



<http://www>

01001011001001001



DATA STRUCTURES USING C++

4th Edition

C M Aslam
Saeed Ahmad
Aqsa Aslam
Atif Mansoor



MAJEED SONS

22 - Urdu Bazar, Lahore. Ph: 37311484

TABLE OF CONTENTS

1	INTRODUCTION TO DATA STRUCTURES & ALGORITHMS	1 – 26
1.1	DATA & INFORMATION.....	1
1.1.1	Data Types.....	1
1.1.2	Abstract Data Types (ADTs).....	2
1.2	DATA STRUCTURES.....	3
1.2.1	Types of Data Structures.....	4
1.2.1.1	Linear Data Structure.....	4
1.2.1.2	Non-Linear Data Structure.....	4
1.2.2	Logical and Physical Data Structures.....	5
1.2.3	Static and Dynamic Data Structures.....	5
1.2.4	Contiguous and Non-Contiguous Data Structures.....	6
1.2.5	Selecting Data Structures.....	6
1.2.6	Operations on Data Structures.....	7
1.2.7	Performance of Data Structures.....	8
1.2.8	Importance of Data Structures.....	8
1.3	ALGORITHM.....	8
1.3.1	Characteristics of an Algorithm.....	9
1.3.2	Types of Algorithms.....	10
1.3.2.1	Linear Algorithms.....	10
1.3.2.2	Divide and Conquer.....	10
1.3.2.3	Greedy Algorithm.....	10
1.3.2.4	Iterative Algorithm.....	11
1.4	WRITING ALGORITHMS.....	11
1.4.1	Algorithmic Notations.....	12
1.4.2	Input & Output Statements.....	15
1.4.3	Selection Statements.....	15
1.4.4	Looping Statements.....	18
1.5	SUB-ALGORITHMS.....	21
1.5.1	Types of Sub-Algorithm.....	21
	Short Answers to the Questions.....	24
	EXERCISES.....	24
2	ALGORITHM ANALYSIS	27 – 31
2.1	ALGORITHM ANALYSIS.....	27
2.1.1	Algorithm Complexity.....	27
2.1.1.1	Space Complexity.....	27
2.1.1.2	Time Complexity.....	27
2.1.2	Order of Growth.....	28

2.2	ASYMPTOTIC ANALYSIS.....
2.2.1	Asymptotic Notations.....
2.2.1.1	Big O-Notation.....
2.2.1.2	Ω -Notation (Big Omega Notation).....
2.2.1.3	Θ -Notation.....
2.2.1.4	Small o-Notation.....
2.2.1.5	ω -Notation.....
2.3	METHODOLOGIES OF ALGORITHM ANALYSIS
2.3.1	Step Counts.....
2.3.2	Solving Recurrences.....
2.3.2.1	Master Method.....
2.3.2.2	Substitution Method.....
2.3.2.3	Recursion Tree Method.....

Answers to the Questions

CISES

ARRAYS	
	51

3.1	INTRODUCTION TO ARRAYS.....
3.1.1	Representation of Array in Memory.....
3.2	OPERATIONS ON LINEAR ARRAYS.....
3.2.1	Traversing Operation.....
2.2.1.1	Complexity Analysis of Traversal Algorithm of Array.....
3.2.2	Insertion Operation.....
3.2.2.1	Inserting Data Item at the End of Array.....
3.2.2.2	Inserting Value at Specified Location in an Array.....
3.2.2.3	Complexity Analysis of Insertion Algorithm of Array.....
3.2.3	Deletion Operation.....
3.2.3.1	Deleting Items at the End.....
3.2.3.2	Deleting Item from a Specified Location.....
3.2.3.3	Complexity Analysis of Deletion Algorithm of Array.....
3.2.4	Search Operation.....
3.2.5	Sorting Operation.....
3.3	MULTI-DIMENSIONAL ARRAYS.....
3.3.1	Complexity Analysis of 2-D Array.....
3.3.2	Higher Dimensional Arrays.....
3.3.3	Algebraic Operations on Matrices.....
3.4	LIMITATIONS OF ARRAYS
3.5	APPLICATIONS OF ARRAYS

Answers to the Questions

CISES

4	STRINGS	75 – 106
4.1	STRINGS	75
4.1.1	Representation of Strings in Memory	75
4.1.2	Types of Strings	75
4.2	STRING OPERATIONS	77
4.2.1	Computing Length of a String	78
4.2.2	Copying Strings	79
4.2.3	String Concatenation	80
4.2.4	Extracting Substring from a String	82
4.2.5	Pattern Matching	83
4.2.5.1	Naive String Matcher	84
4.2.5.2	Rabin-Karp Algorithm	87
4.2.5.3	Knuth-Morris-Pratt Algorithm (KMP)	91
4.2.6	Insertion Operation	95
4.2.7	Deletion Operation	98
4.2.8	Replacement Operation	100
Short Answers to the Questions		105
EXERCISES		105
5	STACKS	107–196
5.1	STACK	107
5.1.1	Implementation of Stack	108
5.1.2	Basic Features of Stack	108
5.1.3	Stack Operations	108
5.1.3.1	PUSH Operation	108
5.1.3.2	POP Operation	109
5.1.3.3	Complexity Analysis of Stack Pushing and Popping	113
5.1.4	Decimal to Binary Conversion using Stack	114
5.1.5	Reversing String using Stack	117
5.2	APPLICATIONS OF STACK	119
5.3	FUNCTION CALL	119
5.3.1	Processor's Stack Operation	121
5.4	EXPRESSIONS	122
5.4.1	Notations used for Expressions	123
5.5	EXPRESSION CONVERSION	124
5.5.1	Infix to Postfix Conversion	124
5.5.2	Infix to Prefix Conversion	13
5.5.3	Postfix to Infix Conversion	14
5.5.4	Prefix to Infix Conversion	15
5.6	EXPRESSION EVALUATION	15
5.6.1	Evaluation of Infix Expression	15

5.6.2	Evaluation of Fully Parenthesized Infix Expression.....	17
5.6.3	Evaluation of Postfix Expression	17
5.6.4	Evaluation of Prefix Expression.....	18
Short Answers to the Questions		19
EXERCISES		19

6	RECURSION	197-202
----------	------------------	----------------

6.1	RECURSION	19
6.1.1	Recursive Function.....	19
6.1.2	Implementation of Recursive Function through Stacks.....	19
6.1.3	Recursion in Math	19
6.1.4	Recursive Programming	19
6.2	Types of Recursion	21
6.2.1	Direct and Indirect Recursions	21
6.2.2	Nested and Non-Nested Recursions	21
6.2.3	Tail and Non-Tail Recursions.....	21
6.2.4	Linear and Tree Recursions.....	21
6.2.5	Excessive Recursion	21
6.3	ADVANTAGES AND DISADVANTAGES OF RECURSION	21
Short Answers to the Questions		21
EXERCISES		21

7	QUEUES	221-228
----------	---------------	----------------

7.1	QUEUES	221
7.1.1	Queue in Programming.....	221
7.1.2	Applications of Queues in Computer System.....	221
7.1.3	Implementation of Queue	221
7.1.4	Basic Features of Queue.....	221
7.1.5	Queue Operations	221
7.1.5.1	Enqueue Operation.....	221
7.1.5.2	Dequeue Operation.....	221
7.1.5.3	Complexity Analysis of Enqueue and Dequeue	221
7.1.6	Drawback of Simple Queue.....	221
7.2	CIRCULAR QUEUE.....	222
7.2.1	Applications of Circular Queue.....	222
7.2.2	Implementation of Circular Queue	222
7.2.3	Circular Queue Operations	222
7.2.3.1	Insertion Operation.....	222
7.2.3.2	Deletion Operation.....	222
7.3	DEQUE	223

7.3.1	Applications of Deque	236
7.3.2	Implementation of Deque	237
7.3.3	Deque Operations	237
7.4	PRIORITY QUEUES	244
7.4.1	Application of Priority Queues in Operating Systems	245
7.4.2	Implementation of Priority Queue	246
Short Answers to the Questions		252
EXERCISES		253

8	SEARCHING & SORTING	257–344
8.1	SEARCHING	257
8.1.1	Sequential Search	257
8.1.1.1	Complexity Analysis of Sequential Search Algorithm	264
8.1.2	Binary Search	265
8.1.2.1	Complexity Analysis of Binary Search Algorithm	269
8.1.3	Hashing	271
8.1.3.1	Hash Table	271
8.1.3.2	Hash Function	271
8.1.3.3	Collisions in Hashing	275
8.1.3.4	Collisions Resolution	275
8.1.3.5	Complexity Analysis of Hash Table	281
8.2	SORTING	282
8.2.1	Bubble Sort	283
8.2.1.1	Complexity Analysis of Bubble Sort Algorithm	287
8.2.2	Selection Sort	288
8.2.2.1	Complexity Analysis of Selection Sort Algorithm	292
8.2.3	Insertion Sort	293
8.2.3.1	Complexity Analysis of Insertion Sort Algorithm	296
8.2.4	Shell Sort	297
8.2.4.1	Complexity Analysis of Shell Sort Algorithm	302
8.2.5	Quick Sort	303
8.2.5.1	Complexity Analysis of Quick Sort Algorithm	309
8.2.6	Merge Sort	311
8.2.6.1	Complexity Analysis of Merge Sort Algorithm	316
8.2.7	Counting Sort	317
8.2.7.1	Complexity Analysis of Counting Sort Algorithm	324
8.2.8	Radix Sort	324
8.2.8.1	Complexity Analysis of Radix Sort Algorithm	33
8.2.9	Bucket Sort	33
8.2.9.1	Complexity Analysis of Bucket Sort Algorithm	34
Short Answers to the Questions		34
EXERCISES		34

9.1	LINKED LISTS	345
9.1.1	Advantages of Linked Lists	346
9.1.2	Disadvantages of Linked Lists	347
9.1.3	Pointer & Linked Lists	347
9.2	SINGLY LINKED LIST	348
9.2.1	Implementation of Singly Linked List	349
9.2.2	Singly Linked List Operations	355
9.2.2.1	Traversing a Singly Linked List	355
9.2.2.2	Searching in Singly Linked List	357
9.2.2.3	Insertion in a Singly Linked List	359
9.2.2.4	Deletion in Singly Linked List	366
9.2.2.5	Sorting in Singly Linked List	371
9.3	DOUBLY LINKED LIST (TWO-WAY LIST)	38
9.3.1	Applications of Doubly Linked List	38
9.3.2	Advantages and Disadvantages of Doubly Linked List	38
9.3.3	Representation of a Doubly Linked List	38
9.3.4	Doubly Linked List Operations	38
9.3.4.1	Traversing Doubly Linked List (DLL)	38
9.3.4.2	Insertion in Doubly Linked List (DLL)	38
9.3.4.3	Deletion in Doubly Linked List	38
9.3.5	Differences between Doubly Linked List and Singly Linked List	4
9.4	CIRCULARLY LINKED LISTS	4
9.4.1	Application of Circular Linked List	
9.4.2	Circularly Double-Linked Lists	
9.5	STACK IMPLEMENTATION USING LINKED LIST	
9.5.1	Operations on Stack using Linked List	
9.6	QUEUE IMPLEMENTATION USING LINKED LIST	
9.6.1	Operations on Queue using Linked List	

Short Answers to the Questions

EXERCISES

10.1	TREE	
10.1.1	Basic Terminologies Related to Trees	
10.1.2	Applications of Tree Data Structure	
10.2	BINARY TREE	
10.2.1	Full Binary Tree	
10.2.2	Perfect Binary Tree	
10.2.3	Complete Binary Tree	
10.2.4	Degenerate Tree	

10.2.5	Skewed Binary Tree	440
10.2.6	Balanced Binary Tree	441
10.2.7	Extended Binary Tree (2-Tree).....	441
10.3	BINARY SEARCH TREE (BST)	442
10.3.1	Representation of Binary Search Tree in Memory	44
10.3.1.1	Linked Storage Technique (Using Linked List).....	44
10.3.1.2	Sequential Storage Technique (Using Array).....	44
10.3.2	Operations on Binary Search Trees	44
10.3.2.1	Search Operation on BST	44
10.3.2.2	Traversal Operation	45
10.3.2.3	Insertion Operation on BST	47
10.3.2.4	Deletion Operation on BST	48
10.4	EXPRESSION BINARY TREE.....	49
10.4.1	Constructing an Expression Binary Tree	4
10.4.2	Constructing Binary Expression Tree using Stack	4
Short Answers to the Questions		4
EXERCISES		4

11	SELF-BALANCING TREES	499-5
11.1	AVL Tree	4
11.1.1	Forms of AVL Trees	
11.1.2	Operations on AVL Tree	
11.1.3	AVL Tree Rotations	
11.1.3.1	Single Rotation.....	
11.1.3.2	Double Rotation	
11.1.4	Insertion Operation in AVL Tree	
11.1.5	Deletion Operation in AVL Tree	
11.1.5.1	Deleting Leaf Node	
11.1.5.2	Deleting Node Having Only One Child	
11.1.5.3	Deleting Node Having Two Children.....	
11.1.6	Complexity Analysis of AVL Tree	
11.2	RED-BLACK TREE.....	
11.2.1	Operations on Red-Black Tree	
11.2.1.1	Insertion in Red-Black Tree	
11.2.1.2	Deletion in Red-Black Tree.....	
11.2.1.3	Comparison of Red-Black Tree with AVL Tree	
11.3	SPLAY TREE	
11.3.1	Splaying.....	
11.3.2	Join Operation in Splay Tree	
11.3.3	Split Operation in Splay Tree	
11.3.4	Insertion Operation in Splay Tree.....	

11.3.5	Deletion Operation in Splay Tree	536
11.4	B-Trees	537
11.4.1	Operations on B-Tree	538
11.4.1.1	Search Operation in B-Tree	539
11.4.1.2	Insertion Operation in B-Tree	539
11.4.1.3	Deletion Operation in B-Tree	542
Short Answers to the Questions		545
EXERCISES		546

12

GRAPHS

549–617

12.1	GRAPHS	549
12.1.1	Basic Terminologies Related to Graphs	550
12.1.2	Applications of Graphs	554
12.2	REPRESENTATION OF GRAPHS	558
12.2.1	Adjacency Matrix Representation	556
12.2.1.1	Representation of Undirected Graph	557
12.2.1.2	Representation of Directed Graph	559
12.2.1.3	Representation of Weighted Graph	560
12.2.1.4	Advantages and Disadvantages of Adjacency Matrix Representation	562
12.2.2	Adjacency List Representation	563
12.2.2.1	Advantages and Disadvantages of Adjacency List Representation	564
12.2.3	Lists vs Matrices	564
12.3	GRAPH TRAVERSAL	565
12.3.1	Breadth-First Search (BFS)	565
12.3.1.1	Applications of Breadth-First Search	568
12.3.2	Depth-First Search (DFS)	569
12.3.2.1	Applications of Depth First Search	574
12.3.3	Difference between BFS & DFS	574
12.4	SPANNING TREE	575
12.4.1	Minimum Spanning Tree (MST)	576
12.4.2	Minimum Spanning Tree Algorithms	577
12.4.2.1	Kruskal's Algorithm	577
12.4.2.2	Prim's Algorithm	581
12.5	SHORTEST PATH PROBLEM	584
12.5.1	Single-Source Shortest Path Problem	586
12.5.1.1	Dijkstra's Algorithm	586
12.5.1.2	Bellman-Ford Algorithm	592
12.5.2	Single-Destination Shortest Path Problem	598
12.5.3	All-Pairs Shortest Path Problem	598

	12.5.3.1 Floyd-Warshall's Algorithm	598
12.6	NP-COMPLETENESS.....	608
	12.6.1 P Problems.....	608
	12.6.2 NP Problems.....	609
	12.6.3 NP-Completeness	610
	Short Answers to the Questions	611
	EXERCISES	612

Other Books of PM SERIES for BCS, MCS, B.Sc

- Operating Systems
- Database Management
- Advanced Programming with C++
- Information Technology
- Visual Basic 6.0
- Web Programming
- Software Engineering
- A Practical Approach to COMPUTER
(with computer concepts and complete latest MS-Office)

Source Code of Programs

Source code of programs given in "DATA STRUCTURES USING C++" can be received by sending e-mail to "pakman_series"

1 DATA & INFORMATION

The word data is the plural of "datum" which means raw facts and figures like numbers, words, images, etc. These numbers, words, and images are discrete and scattered, therefore called facts and figures. Therefore, a collection of raw facts and figures is called *data*. The facts may be about human beings, items, places, natural phenomena, and ideas, etc. Data may be in the form of text, numbers, sounds, images, video clips, etc. A single unit of value in the data is called *data item*. Data are collected for a specific purpose in an organization or system. The processed data that gives useful meaning is called *information*.

In computer memory data is stored as a sequence of 0s and 1s. The '0' and '1' are two binary digits. The binary digit '0' or '1' is called a bit (*bit* is short for *binary digit*). The bits are organized into groups, called *bytes*. One byte consists of 8-bits. The bytes in turn are organized into *words*. These memory units – bits, bytes, and words – are the basic storage structures for different data types (i.e., integer, real numbers, character, and date) in a computer system.

Data in computing (or data processing) are often represented by a combination of items organized in rows and multiple variables organized in columns. Thus in tabular form, the data are organized into fields, records, and files.

The basic terminologies that are used in the data structure are defined as follows:

Entity	Anything in the real world that has a set of different attributes or properties is known as an <i>entity</i> . An entity may be an object with physical existence such as a person, a place, a car, a computer, house, a student, etc. Similarly, an entity may be an object with conceptual existence such as a university course, account, a job, an event, etc.
Field	A single unit of information is called the <i>field</i> . For example, name, address, and date of birth of a student represent fields.
Record	A collection of related fields about an entity is called a <i>record</i> .
File	A collection of a number of records in a system is called a <i>data file</i> or simply <i>file</i> .
Primary Key	A field or set of fields that can uniquely determine a specific record in a database file is called the <i>primary key</i> . The values in the <i>primary key</i> field are called <i>keys</i> or <i>key values</i> .

1.1 Data Type

A data type (in computing terms) is a set of data values having predefined characteristics and a set of operations that can be performed on these data values. Examples of data types are

integer, floats, string, and characters. All programming languages support some data types. The range of values that can be stored in each of these data types are also defined by the programming languages.

Classification of Data Types

Data types can be classified into two types:

- i) Primitive Data Types
- ii) Non-Primitive Data Types

i) Primitive Data Types

The data types that are built-in in a programming language are known as *primitive data types*. These data types are also known as *built-in data types*. The primitive data types are used to represent single values.

Most of the programming languages provide the following built-in data types:

- **Integer:** It is used to represent a number without a decimal point. For example, 12, 45, 90 represent integer values. In C++, int, short, and long data types are used for integer data types.
- **Real:** It is used to represent a number with a decimal point. For example, 1.25, 45.0, 90.01 represent real values. In C++, double, and float data types are used for real data types.
- **Character:** It is used to represent a single character. For example, 'X', '^', 'a', 'A' represent character data. In C++, the *char* data type is used for a character data type.
- **String:** It is used to represent a group of characters. For example, "I Love Pakistan" and "28th of May, 1998" represent string data. This data type is used in some programming languages such as Java, Pascal, and Visual Basic, etc.
- **Boolean:** It is used to represent logical values like *true* and *false* or *yes* and *no*. Almost all programming languages use the Boolean data type.

ii) Non-Primitive Data Types

The data types that are derived from primary or built-in data types are known as *non-primitive data types*. These data types are also known as *derived data types* or *programmatically defined data types*. These data types are used to store a group of values. Examples of non-primitive data types are arrays, structure, union, linked list, stacks, queue, etc. These data types are the building block of complex data structures.

1.1.2 Abstract Data Types (ADTs)

The word 'abstract' means a theoretical or conceptual summary of any object. An abstract data type is a logical description of components of data and the operations that can be performed on them.

performed on data and its implementation details are hidden. The set of operations defines the interface to the ADT. Each operation is defined in terms of its input and output without specifying how the data type is implemented. A *stack* is an example of an ADT, defined in terms of "push" and "pop" operations.

In other words, ADT is a way of looking at a data structure, focusing on what it does, and ignoring how it does. In an ADT, the concept of a certain kind of data structure is separated from the underlying implementation.

For example, integers are ADTs. They are designed as values 0, 1, -1, 2, 3, and by the operations of addition, subtraction, multiplication, division, comparison, etc. Typically, integers are represented in memory as binary numbers, mostly as 2's complement but in some systems as 1's complement. A computer user uses the data as integers and performs operations on this data. He is not concerned about the representation of data inside the computer system (i.e. he/she is independent about the implementation of data in the computer system).

Each ADT consists of the following three parts:

- **Data:** This part describes the structure of the data used in the ADT.
- **Operations:** This part describes valid operations that can be performed on the data and constraints on the operations.
- **Error conditions:** This part describes the error conditions associated with operations.

1.2 DATA STRUCTURES

Information representation has a fundamental significance in computer science. The primary purpose of most computer programs is not only to perform calculations but also to store and retrieve information — usually as fast as possible.

A particular way of storing and organizing data items in a computer's memory or another storage device so that it can be used efficiently is called *data structure*. OR An arrangement of data items in a computer's memory is called a *data structure*. Examples of common data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables.

The representation of a particular data structure in the computer memory is called *storage structure*. There are many storage structures corresponding to particular data structures. The organization of data items on peripheral storage devices such as a disk drive or CD-ROM etc. is called *file structure*.

Data structures provide a means to manage huge amounts of data efficiently such as large databases and Internet indexing services etc. Efficient data structures are key for designing efficient algorithms which are step-wise methods to solve complex problems.

Different kinds of data structures are suited to different kinds of applications. Following are some areas where data structures are applied extensively:

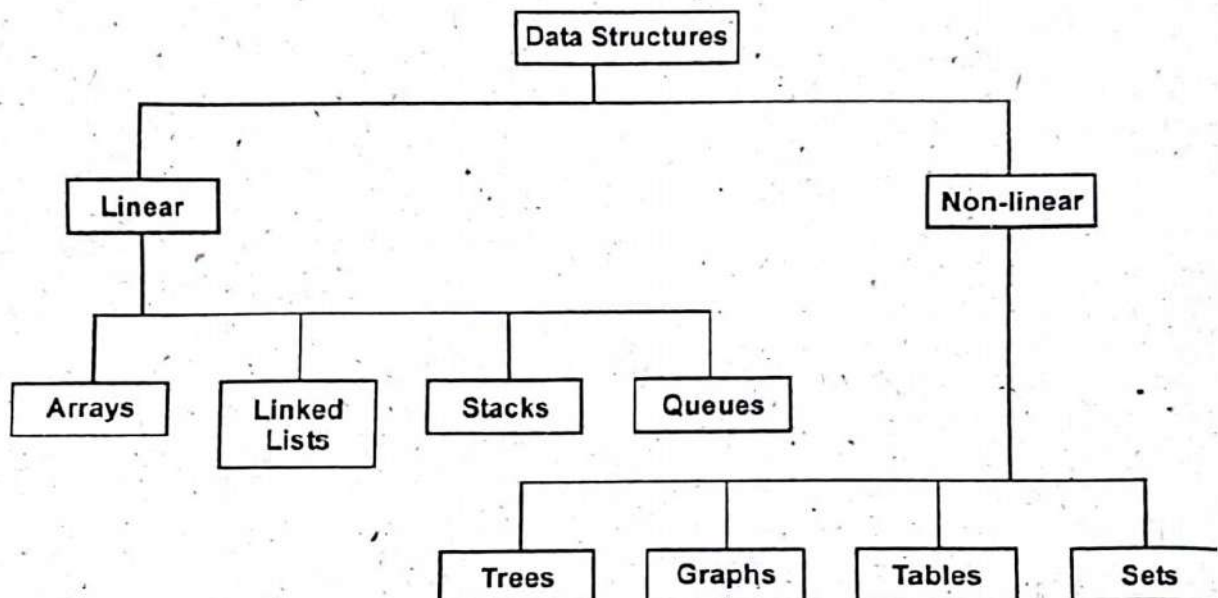
- ★ Compiler Design
- ★ Operating System
- ★ Database Management System

- ★ Statistical analysis package
- ★ Numerical Analysis
- ★ Graphics
- ★ Artificial Intelligence
- ★ Simulations

1.2.1 Types of Data Structures

Following are the major types of data structures:

- i) Linear Data Structure
- ii) Non-Linear Data Structure



1.2.1.1 Linear Data Structure

In a linear data structure, data items are arranged in a sequence or order (or fashion). A linear data structure can be constructed as a continuous arrangement of data elements in the memory. Arrays, linked lists, stacks, and queues are examples of linear structures in which data items are stored in a sequence and can be accessed sequentially.

1.2.1.2 Non-Linear Data Structure

A non-linear data structure is opposite to linear data structure. In this type structure, data items are not arranged in a sequence or order. A non-linear data structure constructed as a collection of a randomly distributed set of data items or elements joined by using a special pointer (or tag). One element of non-linear data structure can be connected more than two adjacent elements. Tree, graph, table, and sets are examples of non-linear structures.

2.2 Logical and Physical Data Structures

Logical data structure describes how the data "seem" to be arranged and the meanings of the data elements in relation to one another. It includes the names of the major elements, their attributes, and the logical relationships between them.

The manner in which data are physically arranged or organized on a storage medium, including various indices and pointers, is called a *physical data structure*. It means that physical data structure concerns to the actual organization of data on a storage device. For example, a data file is a collection of information stored together. This is its logical structure. Physically, however, a file could be stored on a disk in several scattered pieces.

When a data structure is first defined, only the data elements that a user would see, such as a name, address, and cell number, are included. This stage is the logical design. Using the logical design as a basis, the analyst then designs the physical data structures, which include additional elements necessary for implementing the system. Examples of physical design elements are the following:

- **Key Fields:** They are used to locate records in a database table. An example is an item number, which is not required for a business to function but is necessary for identifying and locating specific records in the table.
- **Codes:** They are used to identify the status of master records, such as whether an employee is active (currently employed) or inactive.
- **Transaction Codes:** They are used to identify types of records when a file contains different record types. An example is a credit file containing records for returned items as well as records of payments.
- **Password:** A password is used for accessing the security of a data file.

2.3 Static and Dynamic Data Structures

Static Data Structure

The data structure, in which the number of elements of data is fixed and cannot be changed at run time of the program, is called *static data structure (SDS)*. The size of the static data structure is always known at compilation time. An array is the best example of a static structure.

The static data structure does not mean that we cannot change the assigned values of elements. It is possible to change the contents of a static data structure but cannot increase or decrease the memory space allocated to it.

The use of static data structure is ideal for storing a fixed number of data items. However, it creates many problems and affects the performance of the software. For example, static structures occupy a large size of memory until the termination of the computer program, restricting other programs to execute.

Structure

in which the number of elements of data is not fixed and can be the program, is called *dynamic data structure (DDS)*. The size of the is not known at compilation time, in most programming languages. The or shrink as required by the program at run time. A linked list is an ta structure.

mic data structure is usually stored in non-contiguous memory locations. des more flexibility to adjust the memory consumption as needed by the ther, it provides more efficient methods of insertion and deletion of data

id Non-Contiguous Data Structures

ata Structure

ic in which data is stored in adjacent (contiguous) locations of memory is structure. An array is the best example of a contiguous data structure. ed at contiguous memory locations. In general, contiguous data can be han the data that is stored in fragments because fewer access operations guous memory allocation. Files are sometimes stored in fragments so that used more efficiently (all the small spaces can be used).

us Data Structure

are in which data is stored in different memory locations that are not s) to each other is called a *non-contiguous data structure*. Linked List is ntiguous data structure.

a Structures

ropriate data structure to solve a problem is a very important step in the where to use the array, linked list, etc. First of all, we have to analyze the e resource constraints that a solution must meet. Suppose, the target data in Giga Bytes (GBs) but the available memory space is limited i.e. 200 tern cannot be solved through better programming logic. Otherwise, we e size of memory or buy a new disk with greater space.

necessary to determine the basic operations that must be supported. ource constraints for each operation is needed here. Suppose we have to database and have to search some data items afterward. Let's take the e directory. Suppose there are eight million customer names in the id the information of a particular customer and the information must be able. Selecting suitable data structure for this sort of problem is very entire solution may fail to meet the expectations.

Dynamic Data Structure

The data structure, in which the number of elements of data is not fixed and can be changed at run time of the program, is called *dynamic data structure (DDS)*. The size of the dynamic data structure is not known at compilation time, in most programming languages. The memory size can grow or shrink as required by the program at run time. A linked list is an example of a dynamic data structure.

The data in a dynamic data structure is usually stored in non-contiguous memory locations. This data structure provides more flexibility to adjust the memory consumption as needed by the programs at run time. Further, it provides more efficient methods of insertion and deletion of data elements.

1.2.4 Contiguous and Non-Contiguous Data Structures

Contiguous Data Structure

The data structure in which data is stored in adjacent (contiguous) locations of memory is called a *contiguous data structure*. An array is the best example of a contiguous data structure. Array elements are stored at contiguous memory locations. In general, contiguous data can be accessed more quickly than the data that is stored in fragments because fewer access operations will be required in contiguous memory allocation. Files are sometimes stored in fragments so that the storage space can be used more efficiently (all the small spaces can be used).

Non-Contiguous Data Structure

The data structure in which data is stored in different memory locations that are not adjacent (non-contiguous) to each other is called a *non-contiguous data structure*. Linked List is an example of a non-contiguous data structure.

1.2.5 Selecting Data Structures

Selecting an appropriate data structure to solve a problem is a very important step in the solution. We must know where to use the array, linked list, etc. First of all, we have to analyze the problem to determine the resource constraints that a solution must meet. Suppose, the target data is very large in size i.e. in Giga Bytes (GBs) but the available memory space is limited i.e. 200 Mega Bytes. This problem cannot be solved through better programming logic. Otherwise, we will have to increase the size of memory or buy a new disk with greater space.

Secondly, it is necessary to determine the basic operations that must be supported. Measurement of the resource constraints for each operation is needed here. Suppose we have to insert data in a related database and have to search some data items afterward. Let's take the example of a telephone directory. Suppose there are eight million customer names in the directory. Now, we need the information of a particular customer and the information must be given as early as possible. Selecting suitable data structure for this sort of problem is very important otherwise the entire solution may fail to meet the expectations.

Finally, select the data structure that meets the maximum requirements. Without sufficient experience, it will be difficult to determine which one is the best data structure. We can get help from the Internet, books, or some experienced and related professionals. We may try some existing solutions. After this course, we will be familiar with the data structures and algorithms that are used to solve computer problems.

1.2.6 Operations on Data Structures

Data is useless if it cannot be processed into the required information. We must identify which operations are needed to process the data for solving the given problem. A data structure is created to define and process the data. The data in a data structure is processed using certain operations. Some commonly used operations are as follows:

Insertion	Adding new data items into a data structure is called <i>insertion</i> .
Deletion	Removing data items from a data structure is called a <i>deletion</i> .
Searching	Finding a specific record or data items in a data structure is called <i>searching</i> . Mostly, a specific record is searched based on a given key value. Most of the operations (such as deletion, updating, etc.) are performed on a record or data item after searching it.
Traversing	Accessing each record or item in a data structure exactly once for processing is called <i>traversing</i> . It is also called <i>visiting</i> the data.
Updating	Changing or modifying the data items in the data structure is called <i>updating</i> .
Sorting	Arranging data items in the data structure in a specific order is called <i>sorting</i> .
Merging	Combining multiple groups or lists of data items into a single group or data list is called <i>merging</i> .
Splitting	Dividing a composite data item of a data structure into different pieces of data is called <i>splitting</i> .

The operation through which a data structure is created in memory is called a *creation operation*. Similarly, the operation through which the created data structure is destroyed from memory is called *destruction operation*.

For example, in memory, to create variables 'a' and 'b' of integer type in C++, the statement is written as:

```
int a, b;
```

This statement creates a memory space of two bytes for each variable 'a' and 'b' (total memory space of four bytes).

The operations on data structures are represented directly using the programming language instructions.

1.2.7 Performance of Data Structures

When thinking about a particular application or programming problem, the programmers have to write an algorithm and a program to solve the problem. However, the data structures used for this algorithm or program can greatly affect its performance. A common example is finding an element in a data structure. With an array, this process takes time proportional to the number of elements in the array. With binary search, the search performance will be quick enough. When searching large amounts of data, the other data structure chosen can make a difference in the application's performance that can be visibly measured in seconds.

Since the data structure used by an algorithm can greatly affect the algorithm's performance, there must exist an accurately efficient method to compare the efficiency of various data structures.

1.2.8 Importance of Data Structures

Data structures are important in computer science because these are used in almost every software. A software loads input data in the computer memory for processing. So a specific data structure is essential for the software. Without a data structure, it will be impossible for the software to manage and process data (perform operations of data) inside the computer memory. The data structure also helps to store and organize data on secondary storage devices such as disks, USB drives, memory cards, CDs, etc. Data storage and retrieval on these devices is the need of almost every software. We can say that software engineering greatly depends on the data structure. Complicated programs can be solved easily through smartly selected data structures.

Different kinds of data structures are suited to different computer applications. For example, if we want to store and process data of about 1000 students, we need an array. Some data structures are not used very commonly, like a *Linked List*, *Stack* but they are used very intensively in software like Operating systems, etc. For example, function calls use *Stack* to pass parameters, receive the returned values, and to temporarily store local variables. Similarly, *Stack* is used for undo and redo operations in MS Office applications such as MS-Word, and MS-Excel.

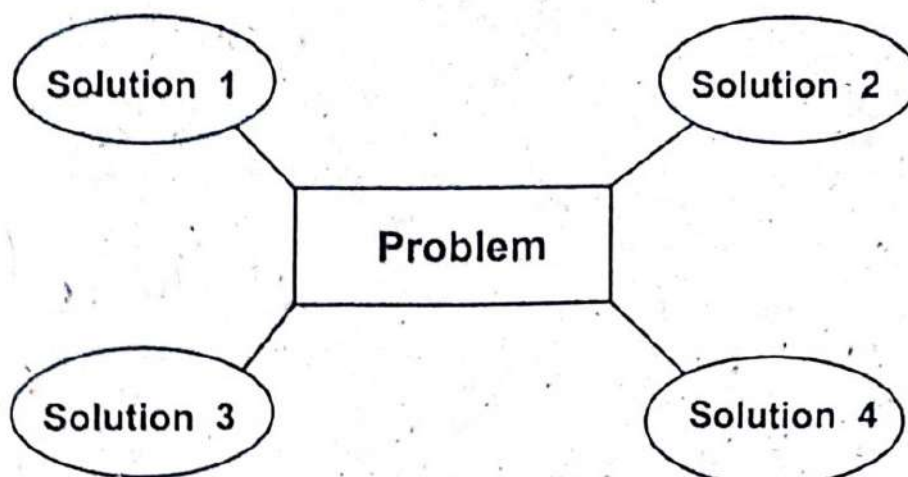
1.3 ALGORITHM

The step-by-step procedure for solving a problem is called an *algorithm*. It also specifies the sequence in which these steps are executed for solving the problem. It terminates after a finite number of steps.

In an algorithm, the problem is divided into a number of simple steps. These steps are arranged in an order. Each step is solved independently. The solution of all steps in an order gives the solution of the problem.

The steps of an algorithm are written in a human-understandable language and are independent of a particular programming language. The algorithmic notations are used to write different steps of the algorithm. The algorithm is the general solution of the problem. After writing the algorithm, it can be implemented in any programming language. It means that a program is written in a programming language by following the algorithm of the problem. Today, C/C++ and Java are the most popular programming languages that are commonly used for implementing algorithms. In this book, C++ is used for implementing the algorithms.

There may be multiple solutions of a given problem. Hence many algorithms can be derived for the solution of a problem. Always the best algorithm is selected for the solution of the problem.



In mathematics, the term algorithm typically refers to a set of steps or formulae used to solve a mathematical computation. For example, step by step procedure is used in long division. FOIL is another example of an algorithm used in algebra. The steps in FOIL are First Outside, Inside Last. This algorithm is useful for multiplying polynomials. BEDMAS is another useful set of steps and is also considered a formula (BEDMAS: Brackets, Exponents, Division, Multiplication, Addition, and Subtraction).

The method for solving problems by dividing them into several simple steps was introduced by a Muslim mathematician "Jafar Muhammad Ben Musa", who was known as "Al-Khuwarizmi". He lived from about 780 to 850. The word 'Algorithm' is also the western adaptation of the word *Al-Khuwarizmi*.

1.3.1 Characteristics of an Algorithm

Please note that any procedure cannot be called an algorithm. An algorithm must have the following characteristics/features:

- **Beginning and End:** An algorithm has a definite beginning and end.
- **Unambiguous:** The word "unambiguous" means "clear-cut" or "definite". An algorithm should be clear and precise. Each step of an algorithm must be clearly defined and must lead to only one meaning.
- **Input:** An algorithm may have many inputs. It takes action on input(s) to achieve the required result.
- **Output:** An algorithm has at least one output (or result). However, it may have more than one output.
- **Finiteness:** An algorithm must terminate after a finite number of steps.
- **Independent:** An algorithm should be independent of any programming language.
- **Feasibility:** An algorithm should be feasible with the available resources.

1.3.2 Types of Algorithms

There are various types of algorithms. Following are some important types of algorithms:

1.3.2.1 Linear Algorithm

A linear algorithm is one of the most basic algorithm to find a particular element in a list of elements. It sequentially checks each element of the list for the target element or value until a match is found or until all the elements have been searched. For example, a linear algorithm is used to count "how many times a specific value exists in a list". Similarly, adding up all the numbers in an array is another example of a linear algorithm.

A linear algorithm is practical when the list has only a few elements, or when performing a single search in an unsorted list. However, a linear algorithm is not practical for performing operations on lists with large elements.

1.3.2.2 Divide and Conquer

The "divide and conquer" is a type of algorithm which divides the problem into sub-problems (pieces) and solves these sub-problems one by one. The solutions of the sub-problems are combined to get the final solution.

The divide and conquer has three parts:

- i) **Divide or Break:** In this step, the problem is divided into smaller sub-problems. Normally, the sub-problems are similar to the original problem.
- ii) **Conquer or Solve:** In this step, the solutions of the sub-problems are solved recursively.
- iii) **Merge or Combine:** In this step, the solutions of the sub-problems are combined to get the solution of the original problem.

For example, the "Merge Sort" algorithm is based on a "divide-and-conquer" programming approach. To sort an array, this algorithm divides the array into two halves, recursively sorts them, and then merges the two sorted halves.

1.3.2.3 Greedy Algorithm

In this type of algorithm, we try to solve problems by selecting the best piece first and then working on the other pieces. For example, to find the largest value in an array, try finding the largest object first and then the smaller one later. Another example of a greedy algorithm is "counting coins". Suppose you are provided coins of Rs. 1, 2, 5, and 10 and you are asked to count the amount of all coins. The greedy procedure will be as follows:

- i) Select Rs. 10 coin.
- ii) Select Rs. 5 coin.
- iii) Select Rs. 2 coin.
- iv) Select Rs. 1 coin.

1.3.2.4 Iterative Algorithm

In this type of algorithm, to obtain the required result, we start with a specific value and repeatedly change it in the direction of the solution. For example, to calculate the factorial of a given number, we start with the number and continue to decrease the number through the algorithm until the number reaches to zero.

1.4 WRITING ALGORITHMS

There is no well-defined standard for writing an algorithm. Rather, it is a problem and resource-dependent. Algorithms are never written to support a particular programming language. All programming languages share basic code constructs like input/output statements, loops, selection statements (flow-control), etc. These common constructs can be used to write an algorithm.

An algorithm is normally written in a step by step manner.

Example:

Following algorithm inputs values in variables A, B, and C. It adds these values and displays result:

```
Step 1-  START
Step 2-  INPUT value in A
Step 3-  INPUT value in B
Step 4-  INPUT value in C
Step 5-  SUM = A + B + C
Step 6-  PRINT SUM
Step 7-  END
```

In writing an algorithm, the use of the word "Step" is optional. So the above algorithm can be written as follows:

```
1-  START
2-  INPUT value in A
3-  INPUT value in B
4-  INPUT value in C
5-  SUM = A + B + C
6-  PRINT SUM
7-  END
```


4.1 Algorithmic Notations

An algorithm consists of several parts and different types of notations are used in these parts. These notations are explained as follows:

- **Name of Algorithm:** Every algorithm is given a name. The name of the algorithm represents the purpose of the algorithm. For example:
 - ★ an algorithm to find the factorial of a number may be given a name "factorial".
 - ★ an algorithm to find the largest number from a list of numbers may be given the name "Find Largest Number".
- **Introductory Comment:** The algorithm name is followed by a brief description of the tasks that the algorithm performs.
- **Steps:** The algorithm consists of a sequence of numbered steps. Each step describes one task to be performed. The instructions in a computer program are written according to these steps.
- **Comments:** Comments are used to explain the purpose of a step or statement. These may be given at the end of each statement or the beginning of the step. In C++, the comments are given starting with double slashes (//). In algorithms, these are written between square brackets [].
- **Variable Name:** A variable is an entity that is used to store a value. The name of the variable is chosen according to the nature of the value it is to hold. For example, a variable that is used to hold the maximum value is named as "Max". Usually, variable names in algorithms are written in capital letters. A variable name consists of letters, numeric digits, and some special characters. It always begins with a letter. Blank spaces are not allowed within variable names.
- **Operators:** Arithmetic (+, -, *, /), relational (<, >, >=, <=, etc.) and logical (AND, OR, NOT) operators are used to describe various mathematical operations.
- **Assignment Statement:** The assignment statement is used to evaluate an expression and assign the calculated value to the variable. The "=" sign is used for the assignment operator. For example, to evaluate the expression $A+B$ and assign its value to a variable S , the assignment statement is written as follows:

$S = A + B$

Algorithm – Sum

Write an algorithm that inputs two numbers and calculates their sum.

-
- 1- START
 - 2- INPUT first number in A
 - 3- INPUT second number in B
 - 4- [Add A to B and store result in S]
 $S = A + B$
 - 5- PRINT S
 - 6- END

Program Code 1.1:

```
// Program to add two numbers
#include <iostream.h>
#include <conio.h>

class add
{
    private:
        int a, b, s;
    public:

        // Member function to input two numbers
        input()
        {
            cout<<"Enter first number : ";
            cin>>a;
            cout<<"Enter second number : ";
            cin>>b;
        } //end of input()

        // Member function to add two numbers and print the result
        print()
        {
            s = a + b;
            cout<<"Sum : "<<s;
        } //end of print()
};

void main(void)
{
    add obj;
    clrscr();
    obj.input();
    obj.print();
    getch();
} //end of main()
```

Output of the Program:

```
Enter first number : 25
Enter second number : 7
Sum = 32
```

Algorithm - Exchange Values

Write an algorithm that exchanges the values of two variables.

- 1- START
- 2- A = 10 [assigns value 10 to variable A]
- 3- B = 20 [assigns value 20 to variable B]

- 4- TEMP = A [assigns the value of A to variable TEMP]
- 5- A = B [assigns the value of B to A]
- 6- B = TEMP [assigns value of TEMP i.e. previous value of A to B]
- 7- END

Program Code 1.2:

```
// Program to swap the values
#include <iostream.h>
#include <conio.h>

class exch
{
    private:
        int A, B, TEMP;
    public:

        // Member function to input two numbers
        input()
        {
            cout<<"Enter first number in A : ";
            cin>>A;
            cout<<"Enter second number in B : ";
            cin>>B;
        } //end of input()

        // Member function to exchange two numbers
        exchange()
        {
            TEMP = A;
            A = B;
            B = TEMP;
            cout<<"\nValues after Exchanging\n\n";
            cout<<"Value in A ="<<A<<endl;
            cout<<"Values in B ="<<B<<endl;
        } //end of exchange()
};

void main(void)
{
    exch obj;
    clrscr();
    obj.input();
    obj.exchange();
    getch();
} //end of main()
```


Output of the Program:

Enter first number in A : 25

Enter second number in B : 78

Values after Exchanging

Value in A =78

Value in B =25

1.4.2 Input & Output Statements

Usually, the INPUT statement is used in algorithms to enter data into a variable. INPUT statement is used with the following format:

INPUT variable_name

For example, to input a value into variable A, the INPUT statement is written as:

INPUT A

Similarly, to print a message or contents of a variable, the PRINT statement is used with the following format:

PRINT message or variable_name

The message is written within double-quotes. To print the contents of a variable, it is written without using double-quotes.

In C++, usually, the "cin" object is used to get input from the keyboard during program execution. The "cout" object is used to print the output on the screen.

1.4.3 Selection Statements

Selection statements are used for making decisions. The decision is made by testing a given condition. After testing a condition, a statement or a set of statements are executed or ignored. The structure that implements this logic in a programming language is called a *conditional structure*. A conditional statement is represented by the IF structure. It has one of the following two forms:

1. IF condition THEN
 Statement(s)
END IF
2. IF condition THEN
 Block A
ELSE
 Block B
END IF

In the first "IF structure", the statements following the "IF structure" are executed when the given condition is true. Otherwise, these statements are ignored.

In the second "IF structure", one of the two blocks of statements is executed. If the given condition is true, then the block of statements under the "IF statement" is executed. Otherwise, the block following the "ELSE statement" is executed.

Algorithm - Greater Number

Write an algorithm that finds the greater of the two given numbers.

- 1- START
- 2- INPUT two numbers in X and Y
- 3- IF $X > Y$ THEN
 PRINT "X is greater"
 ELSE
 PRINT "Y is greater"
 END IF
- 4- END

Program Code 1.3:

// Program to find greater number from the given two numbers

#include <iostream.h>

#include <conio.h>

```
class xy
{
    private:
        int x, y;
    public:

        // Member function to input two numbers
        input()
        {
            cout<<"Enter first number : ";
            cin>>x;
            cout<<"Enter second number : ";
            cin>>y;
        } //end of input()

        // Member function to find greater number
        test()
        {
            if(x>y)
                cout<<"First number is greater";
            else
                cout<<"Second number is greater";
        } //end of test()
}
```



```

};

void main(void)
{
    xy obj;
    clrscr();
    obj.input();
    obj.test();
    getch();
} //end of main()

```

Output of the program:

```

Enter first number : 9
Enter second number : 12
Second number is greater

```

Algorithm – Greater Number

Write an algorithm that finds the largest number from the given three numbers by using nested IF Structure.

- 1- START
- 2- INPUT three numbers X, Y and Z
- 3- IF X > Y THEN
 - IF X > Z THEN
 - PRINT "X is greater"
 - ELSE
 - PRINT "Z is greater"
 - END IF
- ELSE
 - IF Y > Z THEN
 - PRINT "Y is greater"
 - ELSE
 - PRINT "Z is greater"
 - END IF
- END IF
- 4- END

Program Code 1.4:

```

// program that finds the largest number from the given three numbers
#include <iostream.h>
#include <conio.h>

class xyz
{
    private:
        int x, y, z;
    public:
        // Member function to input three numbers

```



```

        input()
        {
            cout<<"Enter first number : ";
            cin>>x;
            cout<<"Enter second number : ";
            cin>>y;
            cout<<"Enter third number : ";
            cin>>z;
        } //end of input()

        // Member function to find the greater number
        test()
        {
            if(x>y)
            {
                if(x>z)
                    cout<<"First number is greater";
                else
                    cout<<"Third number is greater";
            }
            else
            {
                if(y>z)
                    cout<<"Second number is greater";
                else
                    cout<<"Third number is greater";
            }
        } //end of test()
    };

    void main(void)
    {
        xyz obj;
        clrscr();
        obj.input();
        obj.test();
        getch();
    } //end of main()

```

Output of the Program:

```

Enter first number : 12
Enter second number : 5
Enter third number : 8
First number is greater

```

1.4.4 Looping Statements

The looping statements are used to execute certain statements repeatedly. The statements that are executed repeatedly are called the *body of a loop*. In algorithm notation, the "Repeat" statement is used as a looping statement. The "Repeat" statement has two different forms:

- i) REPEAT FOR Loop
- ii) REPEAT WHILE Loop

i) REPEAT FOR Loop

REPEAT FOR loop is used to execute a statement or set of statements for a specific number of times. This loop is also called the *counter loop*. Its general format is as follows:

REPEAT FOR Index = *I-value* TO *F-value* BY *S-value*

Body of loop

[End of loop]

In this syntax:

- I-value** It represents the initial value for the index variable.
- F-value** It represents the final value.
- S-value** It represents the step value i.e. increment/decrement value. Its use is optional. If it is omitted, the value of the index variable is incremented by 1 after each repetition.

The REPEAT FOR loop structure uses an index variable to control the number of iterations of the loop. In the above syntax, 'Index' is the index variable.

Algorithm – Natural Numbers

Write an algorithm that displays the first ten natural numbers using REPEAT FOR Structure.

- 1- START
- 2- REPEAT Step-3 FOR C = 1 To 10 BY 1
- 3- PRINT C
[End of Step-2 Loop]
- 4- END

Program Code 1.5:

// Program displays the first ten natural numbers using *for* loop

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
class xyz
```

```
{
```

```
public:
```

```
// Member function to display natural numbers
```

```
display()
```

```
{
```

```
for(int i = 1; i<=10; i++)
```

```
{
```

```
cout<<i<<endl;
```

```
}
```

```
} //end of display()
```



```

void main(void)
{
    xyz obj;
    clrscr();
    obj.display();
    getch();
} //end of main()

```

ii) REPEAT WHILE Structure

This loop structure is used to execute a statement or a set of statements repeatedly until the given condition remains true. This loop structure is also called a *conditional loop*. Its general syntax is as follows:

REPEAT WHILE(*condition*)

Body of loop

[End of loop]

The condition is tested at the beginning of the loop. There must be a statement in the body of the loop that could change the condition during the execution of the program to bring it near the specified condition.

Algorithm - Natural Numbers

Write an algorithm that displays the first ten natural numbers using the REPEAT WHILE loop structure.

- 1- START
- 2- C = 1
- 3- REPEAT Steps 4 TO 5 WHILE(C <=10)
- 4- PRINT C
- 5- C = C + 1
- [End of Step-3 Loop]
- 6- END

Program Code 1.6:

```

// Program displays the first ten natural numbers using while loop
#include <iostream.h>
#include <conio.h>

class xyz
{
    public:

        // Member function to display natural numbers

```

```

        print()
        {
            int n = 1;
            while(n<=10)
            {
                cout<<n<<endl;
                n = n + 1;
            }
        } //end of print()
    };

void main(void)
{
    xyz obj;
    clrscr();
    obj.print();
    getch();
} //end of main()

```

1.5 SUB-ALGORITHMS

A sub-algorithm is a complete and independently defined algorithmic module. It is called by the main algorithm or by some other sub-algorithm. It can receive values from the calling algorithm. The general form of a sub-algorithm is as follows:

```

Name(p1, p2, ....., pn)
{
    body of sub-algorithm
}

```

Where

Name: It represents the name of the sub algorithm.

p1, p2, ..., pn: They represent the parameters or arguments that receive values from the calling algorithm.

The RETURN statement is used as the last statement of the sub-algorithm to return the control back to the calling algorithm/sub-algorithm.

1.5.1 Types of Sub-Algorithm

The sub-algorithm is divided into two categories: Function sub-algorithm and Procedure sub-algorithm. Both of these types of sub-algorithms are independent modules and are written to perform specific tasks. The differences between the two are as follows:

- ★ The function sub-algorithm returns a single value to the calling algorithm through the RETURN statement. The procedure sub-algorithm may return more than one value. It returns values through parameters.

- ★ The function sub-algorithm is usually called in an expression. The returned value is used in an expression. The procedure sub-algorithm is called by the CALL statement in algorithms.

Function : Mean (X, Y, Z)

Write a function sub-algorithm that finds the average of given three numbers.

MEAN(X, Y, Z)

1. $AVG = (X+Y+Z)/3$
2. Return(AVG)

Program Code 1.7:

```
#include <iostream.h>
#include <conio.h>
```

```
class xyz
```

```
{
```

```
    public:
```

```
        // Member function to find the average of three numbers
```

```
float mean(float x, float y, float z)
```

```
{
```

```
    return (x+y+z)/3;
```

```
}
```

```
};
```

```
void main(void)
```

```
{
```

```
    xyz obj;
```

```
    float a, b, c;
```

```
    clrscr();
```

```
    cout<<"Enter first number : "; cin>>a;
```

```
    cout<<"Enter second number : "; cin>>b;
```

```
    cout<<"Enter third number : "; cin>>c;
```

```
    cout<<"Mean of three numbers is = "<<obj.mean(a,b,c);
```

```
    getch();
```

```
} //end of main()
```

Procedure Algorithm - Exchange Values

Write a procedure sub-algorithm that exchanges the values of two variables.

EXCHANGE(X, Y)

1. TEMP = X
2. X = Y
3. Y = TEMP
4. RETURN

Program Code 1.8:

```
#include <iostream.h>
#include <conio.h>

class xyz
{
    public:
        // Member function to exchange values of two variables
        exchange(int &x, int &y)
        {
            int t;
            t = x;
            x = y;
            y = t;
        } //end of exchange()
};

void main(void)
{
    xyz obj;
    int a, b;
    clrscr();
    cout<<"Enter first number in a : ";
    cin>>a;
    cout<<"Enter second number in b : ";
    cin>>b;
    obj.exchange(a, b);
    cout<<"\nValues after Exchanging\n\n.";
    cout<<"Values in a ="<<a<<endl;
    cout<<"Values in b ="<<b<<endl;
    getch();
} //end of main()
```


Short Answers to the Questions

Q1 Define the data structure. Give examples.

A particular way of storing and organizing data items in a computer's memory or any other storage device so that it can be used efficiently is called *data structure*. OR An arrangement of data items in a computer's memory is called a *data structure*. Examples of common data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables.

Q2 What is the difference between storage structure and file structure?

The representation of a particular data structure in the computer memory is called *storage structure*. The organization of data items on peripheral storage devices such as a disk drive or CD-ROM etc. is called *file structure*.

Q3 What is linear data structure?

In a linear data structure, data items are arranged in a sequence or order (or linear fashion). A linear data structure can be constructed as a continuous arrangement of data items or elements in the memory. Arrays, linked lists, stacks, and queues are examples of linear data structures in which data items are stored in a sequence and can be accessed sequentially.

Q4 What is a non-linear data structure?

A non-linear data structure is opposite to linear data structure. In this type of data structure, data items are not arranged in a sequence or order. A non-linear data structure can be constructed as a collection of a randomly distributed set of data items or elements joined together by using a special pointer (or tag). One element of non-linear data structure can be connected to more than two adjacent elements. Tree, graph, table, and sets are examples of non-linear data structures.

Q5 What is an algorithm?

The step-by-step procedure for solving a problem is called an *algorithm*. It also specifies the sequence in which these steps are executed for solving the problem. It terminates after a finite number of steps.

Q6 What is a linear algorithm?

A linear algorithm is one of the most basic algorithm to find a particular element in a list of elements. It sequentially checks each element of the list for the target element or value until a match is found or until all the elements have been searched. For example, a linear algorithm is used to count "how many times a specific value exists in a list". Similarly, adding up all the numbers in an array is another example of a linear algorithm.

Q7 What is an iterative algorithm?

In this type of algorithm, to obtain the required result, we start with a specific value and repeatedly change it in the direction of the solution. For example, to calculate the factorial of a given number, we start with the number and continue to decrease the number through the algorithm until the number reaches to zero.

EXERCISE

Q.1 Select a correct answer from the multiple choices.

1. A collection of facts and figures is called:
 (a) Information (b) Data (c) Data Structure (d) None
2. A single unit of value in the data is called:
 (a) Datum (b) Record (c) Data Item (d) Data unit
3. A single unit of information is called:
 (a) Datum (b) Record (c) Data Item (d) field
4. Which one is an example of data structure?
 (a) Array (b) Linked List (c) Stack (d) All
5. Which one is an example of linear data structure?
 (a) Array (b) Linked List (c) Trees (d) Both a & b
6. Accessing each item in a data structure exactly once is called:
 (a) Inserting (b) Merging (c) Searching (d) None
7. Combining multiple group or lists of data items into a single group or data list is called:
 (a) Inserting (b) Merging (c) Searching (d) Sorting
8. Which one specifies the sequence in which steps are taken for solving the problem?
 (a) Algorithm (b) Data file (c) Data structure (d) None
9. Which of the following is the type of algorithm?
 (a) Greedy algorithm (b) Divide and Conquer
 (c) Iterative algorithm (d) All
10. How many are there types of sub-algorithms?
 (a) 2 (b) 3 (c) 4 (d) 5

Answers of MCQs

01 (b)	02 (c)	03 (d)	04 (d)	05 (d)	06 (d)	07 (b)	08 (a)	09 (d)	10 (a)
--------	--------	--------	--------	--------	--------	--------	--------	--------	--------

- Q.2 What is data structure? Explain the difference between linear and non-linear data structures.
- Q.3 Describe different data structure operations briefly.
- Q.4 Discuss different methodologies to design an efficient algorithm.

- Q.5** Define the following terms with examples:
- Data Structure
 - Physical Data Structure
 - Logical Data Structure
- Q.6** What is the importance of data structure in computer science.
- Q.7** Differentiate between the following:
- Counter loop & Conditional loop
 - Function sub-algorithm & Procedure sub-algorithm
 - Algorithm & Computer program
 - Physical and logical data structures
- Q.8** Write an algorithm that displays the odd numbers of first N natural numbers.
- Q.9** Write an algorithm that displays the prime numbers of first N natural numbers.
- Q.10** Write a program that finds the largest number from four numbers.
- Q.11** Write a function sub-algorithm to find the exponential power if 'b' represents base and 'p' is its power.
- Q.12** Write an algorithm that displays the sum of even numbers of first N natural numbers.

Source Code of Programs

Source code of programs given in "DATA STRUCTURES USING C++" can be received by sending an e-mail to "pakman_series@hotmail.com".

2.1 ALGORITHM ANALYSIS

Once an algorithm is written for solving a problem and decided to be correct, the next important step is to analyze the algorithm. It is the process of analyzing the problem-solving capability of the algorithm in terms of the running time (or execution time) and memory space required.

Most of the algorithms are designed to work with inputs of arbitrary length. Usually, the running time of an algorithm is stated as a function relating the input length to the number of steps, known as *time complexity*, and size of memory, known as *space complexity*.

Algorithms are often quite different from one another, though the objective of these algorithms is the same. For example, we know that a set of numbers can be sorted using different algorithms. The number of comparisons performed by one algorithm may vary with others for the same input. Hence, the time complexity of those algorithms may differ. At the same time, we need to calculate the memory space required by each algorithm.

The efficiency of an algorithm can be analyzed at two different stages i.e. before and after implementation. These stages are as follows:

- **A Priori Analysis:** This is the theoretical analysis of an algorithm. The efficiency of an algorithm is measured by assuming that all other factors (such as processor speed) are constant and have no effect on the implementation.
- **A Posterior Analysis:** This is the experimental analysis of an algorithm. The selected algorithm is implemented using a programming language. It is then executed on the target computer. In this analysis, the actual running time and space required are collected.

2.1.1 Algorithm Complexity

The complexity of an algorithm is a measurement of the amount of running time and/or storage space required by an algorithm for an input of a given size (n). An algorithm is said to be efficient and fast if it takes less time to execute and consumes less memory space.

Suppose 'X' is an algorithm and 'n' is the size of input data. The running time and memory space required by the algorithm 'X' are the two main factors which decide its efficiency.

- **Time Factor:** The time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor:** Space is measured by counting the maximum memory space required by the algorithm.

Therefore, the performance of an algorithm is measured based on the following properties:

- i) Space Complexity
- ii) Time Complexity

2.1.1.1 Space Complexity

Space Complexity of an algorithm represents the total amount of memory space required by the algorithm to complete its execution. The amount of storage space required by an algorithm varies with the size of the problem to be solved. Space required by an algorithm is equal to the sum of the following:

- A fixed amount of memory space for the program code; simple variables, and constants used in the program. These are independent of the size of the problem being solved.
- A variable amount of memory space required by variables whose size is dependent on the problem being solved. This space increases or decreases if the problem uses dynamic memory allocation, recursion stack space, etc.

Space Complexity $S(P)$ of an algorithm 'P' is $S(P) = C + SP(I)$, where 'C' represents the fixed amount of space and $SP(I)$ represents a variable amount of space of the algorithm which depends on instance characteristic 'I'.

Following simple algorithm explains the concept of space complexity:

Algorithm - SUM(A, B)

- 1- START
- 2- $C = A + B + 10$
- 3- PRINT C
- 4- END

In this algorithm, three variables A, B, C, and one constant value '10' are used. Hence, $S(P) = 1 + 3$. Now space depends on data types of given variables and constant types and it will be multiplied accordingly.

Space Complexity must be taken seriously for multi-user systems and in situations where limited memory is available. Nowadays, Space Complexity is important in the field of embedded computing such as handheld computing-based equipments like cell phones, palm devices, etc.

With the evolution of technology and storage devices, space complexity has been resolved enough. However, the main concern of the analysis of algorithms is the required running time.

2.1.1.2 Time Complexity

Time Complexity of an algorithm represents the amount of time required by the algorithm to complete its execution. Time Complexity is mostly estimated by counting the number of elementary steps or operations performed by the algorithm. However, the running time (or performance) of an algorithm may vary with different types of input data.

Time requirement can be defined as a numerical function $T(n)$, where $T(n)$ can be measured as the number of steps, provided each step consumes constant time. For example the addition of two n -bit integers takes n steps. Consequently, the total computational time is $T(n) = c \cdot n$, where c is the time taken by the addition of two bits. On different computers, the addition of two bits might take different time, say c_1 and c_2 , thus the addition of two n -bit integers takes $T(n) = c_1 \cdot n$ and $T(n) = c_2 \cdot n$ respectively. This shows that different machines result in different slopes, but time $T(n)$ grows linearly as input size increases.

The time complexity of algorithms is most commonly expressed using the big O notation (typically, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^n)$, etc., where n is the input size in units of bits). It is an asymptotic notation to represent the time complexity.

There are various classes of time complexities. Some common time complexities are as follows:

1- Constant Time: $O(1)$

An algorithm is said to run in *constant time* (also written as $O(1)$ time), if it requires the same amount of time regardless of the input size. It means if the value of $T(n)$ is bounded by a value that does not depend on the size of the input. For example, accessing any single element in an array takes constant time as only one operation has to be performed to locate it.

Suppose there is a single statement like this:

```
statement;
```

Time Complexity of this statement will be constant.

2- Linear Time: $O(n)$

An algorithm is said to run in *linear time*, or $O(n)$ time, if its execution time is directly proportional to the input size, i.e. running time grows linearly as input size increases. For example, accessing every element of an array will require running time proportional to the length of an array.

Assume a single statement under a loop:

```
for(i = 0; i < N; i++)
{
    statement;
}
```

Time Complexity for the above statement or algorithm will be linear. The running time of the loop is directly proportional to N . When N doubles, so does the running time.

3- Logarithmic Time: $O(\log n)$

An algorithm is said to run in *logarithmic time* or $O(\log n)$ if its execution time is proportional to the logarithm (i.e. $\log(n)$) of the input size. By default, the base of $\log$ is 2. Every time the size of the input doubles, the algorithm performs one more step. For example, a binary search algorithm takes logarithmic time.

Consider the following code:

```
while(low <= high)
{
    mid = (low + high) / 2;
    if (target < list[mid])
        high = mid - 1;
    else if (target > list[mid])
        low = mid + 1;
    else break;
}
```

This code divides the working area in half with each iteration. Now, this code or algorithm will have a Logarithmic Time Complexity. The running time of the algorithm is proportional to the number of times N can be divided by 2.

An $O(\log n)$ algorithm is considered highly efficient, as the ratio of the number of operations to the size of the input decreases and tends to zero when ' n ' increases. An algorithm that must access all elements of its input cannot take logarithmic time, as the time taken for reading input of size ' n ' is of the order of ' n '.

4- Log-Linear Time: $O(n \log n)$

An algorithm is said to run in a *log-linear time* when it calls another *logarithmic time algorithm* inside a loop. Log-linear time is also known as *quasi-linear time*. For example, the merge sort algorithm takes log-linear time.

5- Quadratic Time: $O(n^2)$

An algorithm is said to run in *quadratic time* or $O(n^2)$ if its execution time is proportional to the square of the input size. Algorithms taking quadratic time are used by bubble sort, selection sort, and insertion sort.

Consider the following nested loop:

```
for(i = 0; i < N; i++)
{
    for(j = 0; j < N; j++)
```



```

    {
        statement;
    }
}

```

In this case, the Time Complexity for the above code will be Quadratic. The running time for two loops is proportional to the square of N . When N doubles, the running time increases by 4.

Cubic Time: $O(n^3)$

An algorithm is said to run in *cubic time* or $O(n^3)$, if its execution time is proportional to the cube of 'n' of the input size. In other words, if the input doubles, the number of steps increases by 8.

Consider the following three loops in nested form:

```

for ( i = 0; i < N; i++ )
{
    for ( j = 0; j < N; j++ )
    {
        for ( k = 0; k < N; k++ )
            statement;
    }
}

```

In this case, the Time Complexity for the above code will be Cubic. The running time for three loops is proportional to the cube of N . When ' N ' triples, the running time increases by 27.

Polynomial Time: $O(n^k)$

An algorithm is said to run in *polynomial time* if the number of steps required to complete the algorithm for a given input is $O(n^k)$ for some non-negative integer ' k ', where ' n ' is the size of the input. Polynomial-time algorithms are said to be "fast". For example, a sorting algorithm takes polynomial time. Similarly, all basic arithmetic operations such as addition, subtraction, multiplication, and division, as well as computing square roots, exponentiation, and logarithms, can be performed in polynomial time.

Suppose we have a list of size ' N ' and we have nested loops on that list such as:

```

for ( i = 0; i < N; i++ )
    for ( j = 0; j < N; j++ )
        Some processing on the list

```

In the above code, the processing part is executed $N*N$ times i.e. N^2 times. Such a code has $O(N^2)$ time complexity.

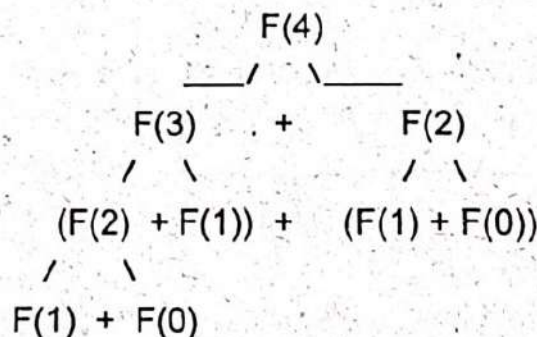
8- Exponential Time: $O(2^n)$

An algorithm is said to run in *exponential time* or $O(2^n)$, if it takes time proportional to 2^n . It performs calculations double as the input grows. Most of the recursion functions are performed in exponential time.

Consider the following code for recursion function to find N^{th} Fibonacci series:

```
int F(n)
{
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return F(n-1) + F(n-2);
}
```

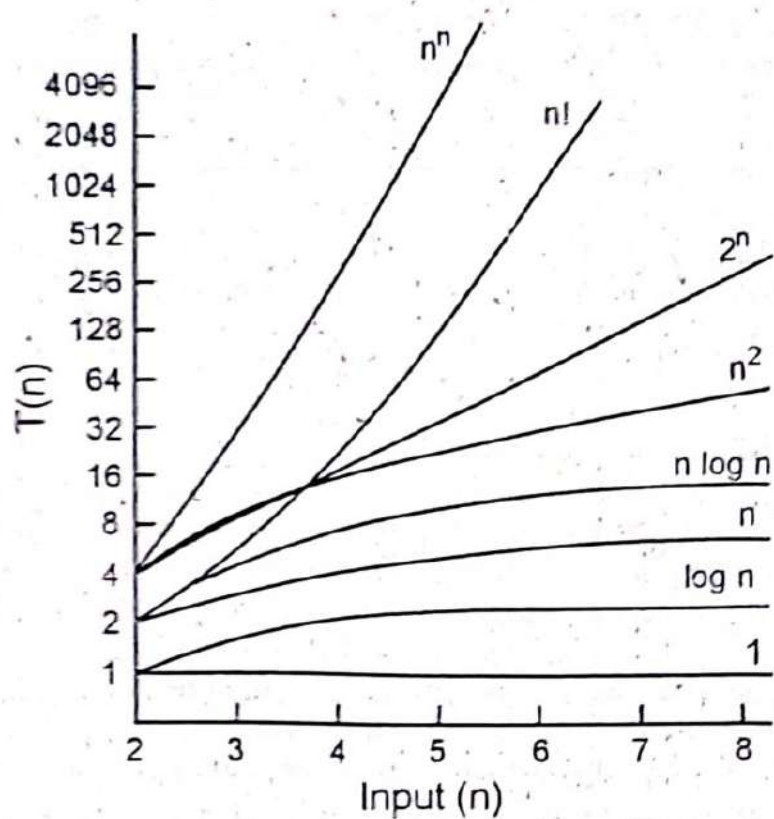
In the above code, for every call to $F()$, we make two more calls to $F()$ in the else part. So if we call $F(4)$, the tree of calls to $F()$ would look like:



This tree keeps growing exponentially as we increase N . Hence the Time Complexity will be $O(2^N)$.

2.1.2 Order of Growth

The order of growth means how the performance of the algorithm is going to increase or decrease by increasing the input size. For example, $O(1)$ remains constant on increasing or decreasing input size. But $O(n^2)$ increases as the input size increases. The order of growth for various classes of time complexities are shown in the following graph:



Class Activity

Create a table for values of n [1 – 30]. Compute $\log n$, $n \log n$, n^2 , 2^n , $n!$ and n^n for each value of ' n ' and fill in the table like:

N	1	2	3	4	5	6	...
$\log n$							
$n \log n$							
n^2							
2^n							
$n!$							
n^n							

Then plot a graph for the created table and compare it with the figure above.

The following table shows the order of growth:

Functions	Name
C	Constant
$\log n$	Logarithmic
$\log^2 n$	Log-squared
N	Linear
$n \log n$	Log-linear

n^2	Quadratic
n^3	Cubic
2^n	Exponential

In mathematics, the addition rule is: " $n + n = 2n$ " but for analysis of the algorithm, if the complexity of an algorithm is " $n + n^2$ ", then it is supposed to choose the highest growth function which is, in this case, is " n^2 ".

2.2 ASYMPTOTIC ANALYSIS

In mathematical analysis, asymptotic analysis is also known as *asymptotics*. It is a method of describing limiting behavior i.e. upper bound, lower bound, or average. Asymptotic analysis of an algorithm refers to computing the execution time (or running time) of any operation in mathematical units of computation. For example, the running time of one operation is computed as $f(n)$ and may be for another operation it is computed as $g(n^2)$. This means that the running time of the first operation will increase linearly with the increase in n and the running time of the second operation will increase exponentially when n increases. Similarly, the running time of both operations will be nearly the same if n is significantly small.

Usually, the time required by an algorithm falls under three types of analysis:

- i) Worst-Case
- ii) Best-Case
- iii) Average-Case

i) Worst-Case

The maximum (or upper bound) execution time required by an algorithm to complete the task is called *worst-case*. In the worst-case analysis, we calculate the maximum number of steps or operations required by the algorithm on the input data to complete a certain task. For example, in the linear search problem, the worst-case happens when the element to be searched is not present inside the list i.e. an array. Therefore, the algorithm will compare all the elements of the array one by one.

In most of the algorithm analysis, we are mostly concerned with worst-case analysis. It is most excellent than other cases because we can guarantee for required maximum time to complete a certain task by an algorithm. In real-time computing, the worst-case execution time of an algorithm is mostly concerned.

ii) Best-Case

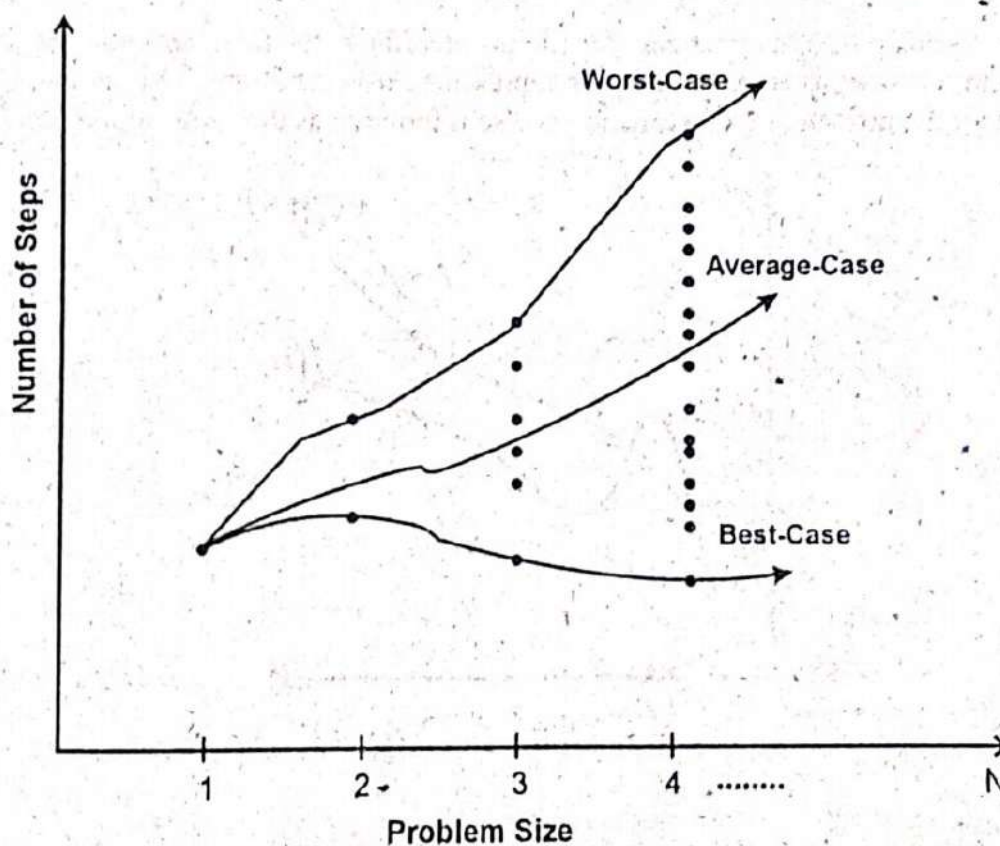
The minimum (or lower bound) execution time required by an algorithm to complete the task is called *best-case*. In the best-case analysis, we calculate the minimum number of steps or operations required by the algorithm on the input data to complete a certain task. For example, in the linear search problem, the best-case happens when the element to be searched is present at the first location.

The best-case analysis is bogus because it gives an idea for the lowest possible time to complete a task. It does not guarantee us how much maximum time required to complete a certain task of an algorithm.

III) Average-Case

The average execution time required by an algorithm to complete the task is called *average-case*. In the average-case analysis, we calculate the number of steps or operations required by the algorithm on the input data to complete a certain task. We take all possible inputs and calculate the computing time for all the inputs. Then, we sum all the calculated values and divide the sum by the total number of inputs. For example, in the linear search problem, we sum all the cases and divide the sum by $(n+1)$.

The average case analysis is not easy to do in most of the practical cases and it is rarely used. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.



2.2.1 Asymptotic Notations

Asymptotic notations are one of the ways to represent the running time of algorithms. Sometimes we are interested in the *worst-case* running time only. Similarly, sometimes we wish to cover all inputs, not just the *worst-case*. Asymptotic notations are well suited to characterizing running time no matter what the input.

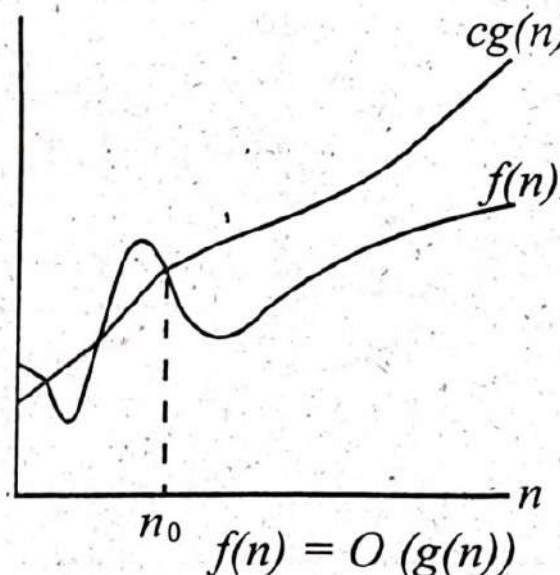
Following are the commonly used asymptotic notations to represent or express the running time complexity of algorithms:

- i) Big O-Notation
- ii) Ω -Notation
- ii) Θ -Notation
- iv) Small o-Notation
- v) ω -Notation

2.2.1.1 Big O-Notation

Big O-notation is a mathematical notation. It describes the limiting behavior of a function when the argument tends towards a particular value or infinity. In computer science, it is used to express the *upper bound* of an algorithm's running time. It is specifically used to measure the *worst-case* time complexity of an algorithm. It describes how the running time of an algorithm grows relative to the input, as the input gets larger.

Big O-notation characterizes functions according to their growth rates. Different functions with the same growth rate may be represented using the same O-notation. The letter O is used because the growth rate of a function is also referred to as the *order of the function*.



For a given function $g(n)$, it is denoted as $O(g(n))$, and pronounced as 'big-oh of g of n'. It is expressed mathematically as:

$$O(g(n)) = \{f(n): \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

Graphically it is shown as:

2.2.1.2 Ω -Notation (Big Omega Notation)

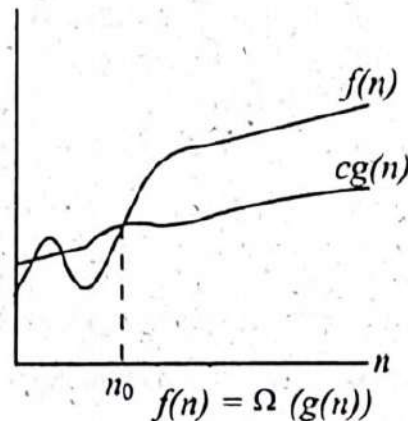
Ω -notation is used to express the *lower bound* of an algorithm's running time. It is specifically used to measure the *best-case* time complexity of an algorithm.

For a given function $g(n)$, it is denoted as $\Omega(g(n))$, and pronounced as 'big-omega of g of n '. It is expressed mathematically as:

$$\Omega(g(n)) = \{f(n): \text{there exists positive constants } c \text{ and } n_0 \text{ such that}$$

$$0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

Graphically it is shown as:



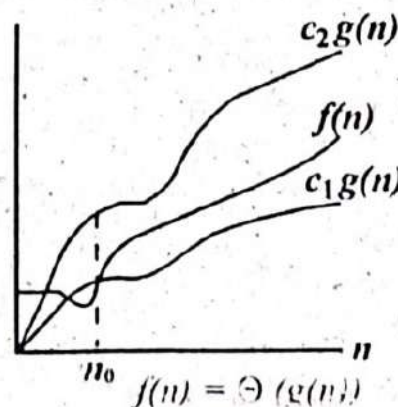
2.2.1.3 Θ -Notation

Θ -notation is used to express both the *lower bound* and *upper bound* of an algorithm's running time. For a given function $g(n)$, it is denoted as $\Theta(g(n))$, and pronounced as 'big-theta, of g of n '. It is expressed mathematically as:

$$\Theta(g(n)) = \{f(n): \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that}$$

$$0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\}$$

The above definition means, if $f(n)$ is theta of $g(n)$, then the value $f(n)$ is always between $c_1g(n)$ and $c_2g(n)$ for large values of n ($n \geq n_0$). The definition of theta also requires that $f(n)$ must be non-negative for values of ' n ' greater than ' n_0 '. It is graphically represented as:



2.2.1.4 Small o-Notation

Small o-notation is same as big O-notation except that big-O is inclusive upper bound, while small o-notation is a strict upper bound. For a given function $g(n)$, it is denoted as $o(g(n))$ and pronounced as 'small-o, of g of n'. It is expressed mathematically as:

$$o(g(n)) = \{f(n): \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}$$

2.2.1.5 ω - Notation

ω - notation is same as Ω notation except that Ω is inclusive lower bound, while ω is a strict lower bound. For a given function $g(n)$, it is denoted as $\omega(g(n))$ and pronounced as 'small-omega, of g of n'. It is expressed mathematically as:

$$\omega(g(n)) = \{f(n): \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}$$

2.3 METHODOLOGIES OF ALGORITHM ANALYSIS

Algorithms can be analyzed using the following methodologies based on their nature:

- 1) Step Counts
- 2) Solution to Recurrences

2.3.1 Step Counts

In this method, the running time of each step is calculated and the highest of the running time among all steps is the overall running time of an algorithm. We typically ignore small values, since we are usually interested in estimating how slow the program will be on large inputs.

Example:

Let us calculate the running time of an algorithm for all three cases: best-case, worst-case, and average-case.

i) Best-Case

The algorithm with its cost and running time for each step is as follows (each operation has a cost (or weight) and is represented by c_n):

	Cost	Running Time
for(int i=1; i<n; i++)	C_1	n

{		
int j = i;	C_2	$n-1$
while(j > 0 && a[j-1] > a[j])	C_3	$n-1$
{		
int temp = a[j];	C_4	0
a[j] = a[j-1];	C_5	0
a[j-1] = temp;	C_6	0
j--;	C_7	0
}		
}		

The above-given algorithm does the sorting operation on a given set of inputs. For best-case, consider the input is already sorted, then the inner while loop of the algorithm will never be executed. The running time for the best-case of the algorithm is as follows:

$$T(n) = c_1(n) + c_2(n-1) + c_3(n-1) + c_4(0) + c_5(0) + c_6(0) + c_7(0)$$

$$T(n) = nc_1 + (n-1)c_2 + (n-1)c_3$$

$$T(n) = nc_1 + nc_2 - c_2 + nc_3 - c_3$$

$$T(n) = n(c_1 + c_2 + c_3) - c_2 - c_3$$

Let $c_1 + c_2 + c_3 = a$ and $c_2 + c_3 = b$, then

$$T(n) = an - b$$

In order of growth, 'n' has the highest order than the constant. Hence, the running time for the best-case of the algorithm is 'n'.

ii) Worst-Case

The algorithm with its cost and running time of each step for worst-case is as follows:

	Cost	Running Time
for(int i = 1; i < n; i++)	C_1	n
{		
int j = i;	C_2	$n-1$
while(j > 0 && a[j-1] > a[j])	C_3	$\sum_{j=2}^n t_j - 1$
{		
int temp = a[j];	C_4	$\sum_{j=2}^n t_{(j-1)}$

$a[j] = a[j-1];$	C_5	$\sum_{j=2}^n t_{(j-1)}$
$a[j-1] = temp;$	C_6	$\sum_{j=2}^n t_{(j-1)}$
$j--;$	C_7	$\sum_{j=2}^n t_{(j-1)}$
$\}$		
$\}$		

The above given running time of the algorithm is for worst-case. Consider the input is sorted in the opposite, then while loop is always true for each iteration. Hence, the running time for while loop will be $\sum_{j=2}^n t_j$ (number of times while loop will be executed) minus one as it is the inner loop of the outer loop. If these running times are expanded using arithmetic series then:

$$\sum_{j=2}^n t_j = \frac{n(n+1)}{2}$$

and

$$\sum_{j=2}^n t_{(j-1)} = \frac{(n-1)(n-1+1)}{2} = \frac{n(n-1)}{2}$$

The running time for the worst-case of algorithm is as follows:

$$T(n) = c_1(n) + c_2(n-1) + c_3 \left(\sum_{j=2}^n t_j - 1 \right) + c_4 \left(\sum_{j=2}^n t_{(j-1)} \right) + c_5 \left(\sum_{j=2}^n t_{(j-1)} \right) + c_6 \left(\sum_{j=2}^n t_{(j-1)} \right) + c_7 \left(\sum_{j=2}^n t_{(j-1)} \right)$$

$$T(n) = c_1(n) + c_2(n-1) + c_3 \left(\frac{n(n+1)}{2} - 1 \right) + c_4 \left(\frac{n(n-1)}{2} \right) + c_5 \left(\frac{n(n-1)}{2} \right) + c_6 \left(\frac{n(n-1)}{2} \right) + c_7 \left(\frac{n(n-1)}{2} \right)$$

$$T(n) = c_1(n) + c_2(n-1) + c_3 \left(\frac{n^2+n}{2} - 1 \right) + c_4 \left(\frac{n^2-n}{2} \right) + c_5 \left(\frac{n^2-n}{2} \right) + c_6 \left(\frac{n^2-n}{2} \right) + c_7 \left(\frac{n^2-n}{2} \right)$$

$$T(n) = c_1(n) + c_2(n-1) + c_3 \left(\frac{n^2+n-2}{2} \right) + c_4 \left(\frac{n^2-n}{2} \right) + c_5 \left(\frac{n^2-n}{2} \right)$$

$$+ c_6 \left(\frac{n^2 - n}{2} \right) + c_7 \left(\frac{n^2 - n}{2} \right)$$

$$T(n) = c_1 n + c_2 n - c_2 + \frac{c_3 n^2}{2} + \frac{c_3 n}{2} - c_3 + \frac{c_4 n^2}{2} - \frac{c_4 n}{2} + \frac{c_5 n^2}{2} - \frac{c_5 n}{2} + \frac{c_6 n^2}{2} - \frac{c_6 n}{2} + \frac{c_7 n^2}{2} - \frac{c_7 n}{2}$$

$$T(n) = n^2 \left[\frac{c_3 + c_4 + c_5 + c_6 + c_7}{2} \right] + n \left[c_1 + c_2 + \frac{c_3}{2} - \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} \right] - c_2 - c_3$$

Let $\frac{c_3 + c_4 + c_5 + c_6 + c_7}{2} = a$, $c_1 + c_2 + \frac{c_3}{2} - \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} = b$, and $c_2 + c_3 = c$, then

$$T(n) = an^2 + bn - c$$

In order of growth, n^2 has the highest order. Hence, the running time for the worst-case of the algorithm is n^2 .

For the average case, the running time for the while loop will be: $\sum_{j=2}^n \frac{j}{2} - 1$, and the statements within while will be: $\sum_{j=2}^n \frac{(j-1)}{2}$.

Example 1:

Consider the following code:

```
int count = 0;
for (int i = 0; i < n; i++)
    for (int j = 0; j < i; j++)
        count++;
```

The iterations will be followed as:

- When $i = 0$, it will run 0 time.
- When $i = 1$, it will run 1 time.
- When $i = 2$, it will run 2 times and so on.

Total number of times count++ will run is: $0 + 1 + 2 + \dots + (n-1) = \frac{n(n-1)}{2}$. So, the time complexity will be $O(n^2)$.

Example 2:

Consider the following code:

```
int count = 0;
```

```

for (int i = n; i > 0; i/=2)
    for (int j = 0; j < i; j++)
        count++;

```

The iterations for count++ will be followed as:

- When $i = n$, it will run ' n ' times.
- When $i = \frac{n}{2}$, it will run ' $\frac{n}{2}$ ' times.
- When $i = \frac{n}{4}$, it will run ' $\frac{n}{4}$ ' times and so on.

Total number of times count++ will run is $n + \frac{n}{2} + \frac{n}{4} + \dots + 1 = 2n$. So, the time complexity will be $O(n)$.

2.3.2 Solving Recurrences

A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs. Recurrence equations arise frequently in the analysis of algorithms, particularly in the analysis of recursive as well as divide-and-conquer algorithms. For example, the worst-case running time of merge sort is described by recurrence as follows:

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \theta(n) & \text{if } n > 1 \end{cases}$$

Whose solution claimed to be: $T(n) = \theta(n \log n)$. Recurrences can take many forms. They might divide sub-problems into equal sizes or might not. Three methods to solve recurrences for obtaining asymptotic θ or O bounds on the solution are as follows:

- i) Master method
- ii) Substitution method
- iii) Recursion Tree method

2.3.2.1 Master Method

Master method is the easiest way for obtaining the runtime of a recursive algorithm. It applies only on the following format of function:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \quad \text{for } a \geq 1, \text{ } b > 1$$

Where 'a' depicts the number of sub-problems, and 'b' depicts the size of each sub-problem. For example, a binary search has a '1' split. It means the value of 'a' for binary sort is '1' and it cuts the input in half. Therefore, for binary search value of 'b' will be $\frac{n}{2}$.

Once a , b , and $f(n)$ are determined, calculate the work done by recursion using $O(n^{\log_b a})$, and compare it with the following cases of Master theorem or method.

CASE-I:

If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then

$$T(n) = \theta(n^{\log_b a})$$

CASE-II:

If $f(n) = \theta(n^{\log_b a})$, then

$$T(n) = \theta(n^{\log_b a} \log n)$$

CASE-III:

If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, then check regularity condition.

Regularity Condition: If $af\left(\frac{n}{b}\right) \leq c \cdot f(n)$ for some constant $c < 1$, then

$$T(n) = \theta(f(n))$$

Examples:

- 1- $9T\left(\frac{n}{3}\right) + n$, here $a = 9$, $b = 3$, and $f(n) = n$.

Calculate $O(n^{\log_b a})$, we have

$$n^{\log_b a} = n^{\log_3 9} = n^{\log_3 (3)^2} = n^{2\log_3 (3)} = n^2$$

It satisfies the case-I by comparing $O(n^{\log_b a})$ with $f(n)$ i.e. $n^2 > n$.

$$\text{Hence, } T(n) = \theta(n^2).$$

- 2- $9T\left(\frac{n}{3}\right) + n^3$, here $a = 9$, $b = 3$, and $f(n) = n^3$

Calculate $O(n^{\log_b a})$, we have

$$n^{\log_b a} = n^{\log_3 9} = n^{\log_3 (3)^2} = n^{2\log_3 (3)} = n^2$$

It satisfies the first condition of Case-III by comparing $O(n^{\log_b a})$ with $f(n)$ i.e. $n^2 < n^3$. Now check regularity condition:

$$af\left(\frac{n}{b}\right) \leq c \cdot f(n)$$

$$9\left(\frac{n}{3}\right)^3 \leq c \cdot n^3$$

$$9 \frac{n^3}{27} \leq c \cdot n^3$$

$$\frac{n^3}{3} \leq c \cdot n^3$$

$$\frac{1}{3} \leq c$$

Regularity condition also satisfied. Hence, apply Case-III for results.

$$T(n) = \theta(n^3)$$

3- $2T\left(\frac{n}{4}\right) + 1$, here $a = 2$, $b = 4$, and $f(n) = 1$

Calculate $O(n^{\log_b a})$, we have

$$n^{\log_b a} = n^{\log_4 2} = n^{\frac{1}{2} \log_2 2} = n^{\frac{1}{2}} = \sqrt{n}$$

It satisfies the condition of case-I i.e. $O(\sqrt{n}) > O(1)$. Thus, the running time is

$$T(n) = \theta(\sqrt{n})$$

Master Method does not support the following formats:

- ❖ If $T(n)$ is not monotone e.g. $T(n) = \sin(n)$.
- ❖ If $f(n)$ is not polynomial e.g. $T(n) = 2T\left(\frac{n}{2}\right) + 2^n$
- ❖ If b is not expressed as a constant e.g. $T(n) = 2T(\sqrt{n}) + n$

For these cases, one should switch over to the substitution method or recursion method.

2.3.2.2 Substitution Method

In this method, a bound is guessed and a mathematical induction technique is used to prove that the assumption is correct. It has two steps:

- i) Guess the form of the solution.
- ii) Use the mathematical induction to find the constants and show that the solution works.

Examples:

1- $2T\left(\frac{n}{2}\right) + n$

Guess of the solution be: $T(n) = n \log n$

Substitute this guess to the recurrence equation, we have;

$$T(n) = 2 \left[\frac{n}{2} \log \left(\frac{n}{2} \right) \right] + n$$

$$T(n) = 2 \left[\frac{n}{2} (\log n - \log 2) \right] + n$$

$$T(n) = 2 \left[\frac{n}{2} (\log n - 1) \right] + n \quad \{ \text{as } \log 2 = 1 \}$$

$$T(n) = [n \log n - n] + n$$

$$T(n) = n \cdot \log n$$

Which is equal to the guess. Hence, proved that the assumption is correct for this equation.

2- $T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$

Let the guess be: $T(n) = n \log_{3/2} n$

Substitute this guess to the recurrence equation, we have:

$$T(n) = \left[\frac{n}{3} \log_{3/2} \frac{n}{3} \right] + \left[\frac{2n}{3} \log_{3/2} \frac{2n}{3} \right] + n$$

$$T(n) = \left[\frac{n}{3} (\log_{3/2} n - \log_{3/2} 3) \right] + \left[\frac{2n}{3} (\log_{3/2} n - \log_{3/2} \frac{2}{3}) \right] + n$$

We know $\log_{3/2} \left(\frac{2}{3}\right) = -1$, and let $\log_{3/2}(3) = c$, then

$$T(n) = \left[\frac{n}{3} (\log_{3/2} n - c) \right] + \left[\frac{2n}{3} (\log_{3/2} n + 1) \right] + n$$

$$T(n) = \frac{n}{3} \log_{3/2} n - \frac{nc}{3} + \frac{2n}{3} \log_{3/2} n + \frac{2n}{3} + n$$

$$T(n) = \frac{n}{3} \log_{3/2} n + \frac{2n}{3} \log_{3/2} n + \frac{2n}{3} + n - \frac{nc}{3}$$

$$T(n) = \frac{n}{3} \log_{3/2} n + \frac{2n}{3} \log_{3/2} n + \left(\frac{2}{3} + 1 - \frac{c}{3} \right) n$$

Let $\frac{2}{3} + 1 - \frac{c}{3} = k$, then substitute it to the above equation to simplify it.

$$T(n) = \frac{n}{3} \log_{3/2} n + \frac{2n}{3} \log_{3/2} n + kn$$

$$T(n) = \frac{3n}{3} \log_{3/2} n + kn$$

$$T(n) = n \log_{3/2} n + kn$$

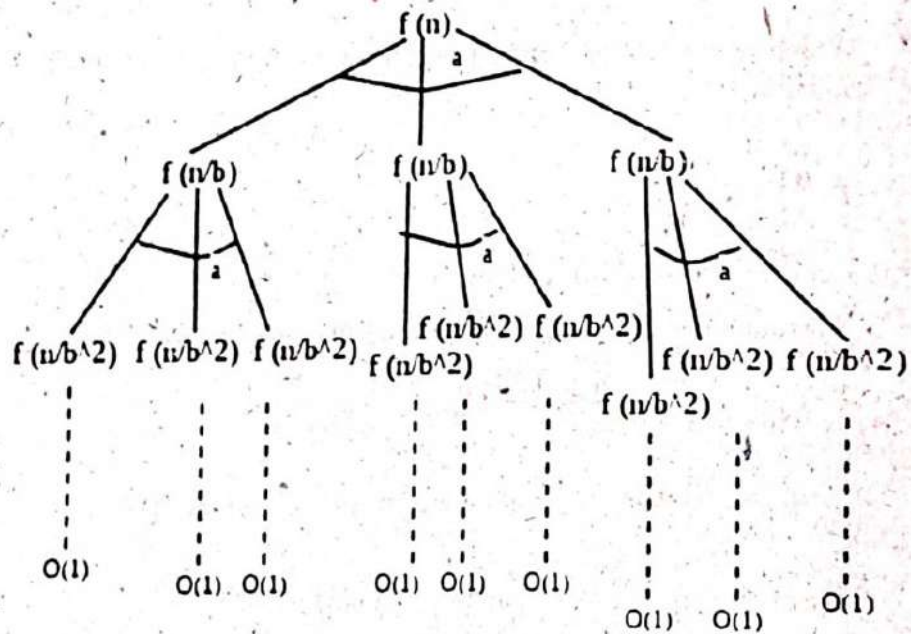
Because $n \log_{3/2} n$ has highest growth than kn . Thus,

$$T(n) = n \log_{3/2} n$$

Which is equal to the guess. Hence, proved that the assumption is correct for this equation.

2.3.2.3 Recursion Tree Method

Recursion Tree method provides a good guess for recurrence. It converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. Then up the numbers in each node to get the entire cost of the algorithm. It takes a form as:

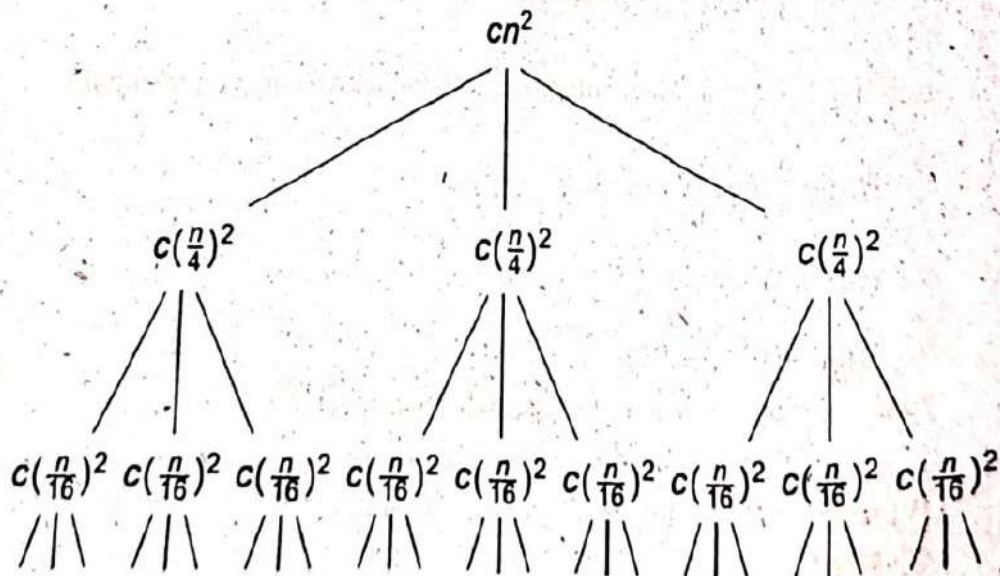


The root node has $f(n)$ function. The height of the recurrence tree is $\log_b n$.

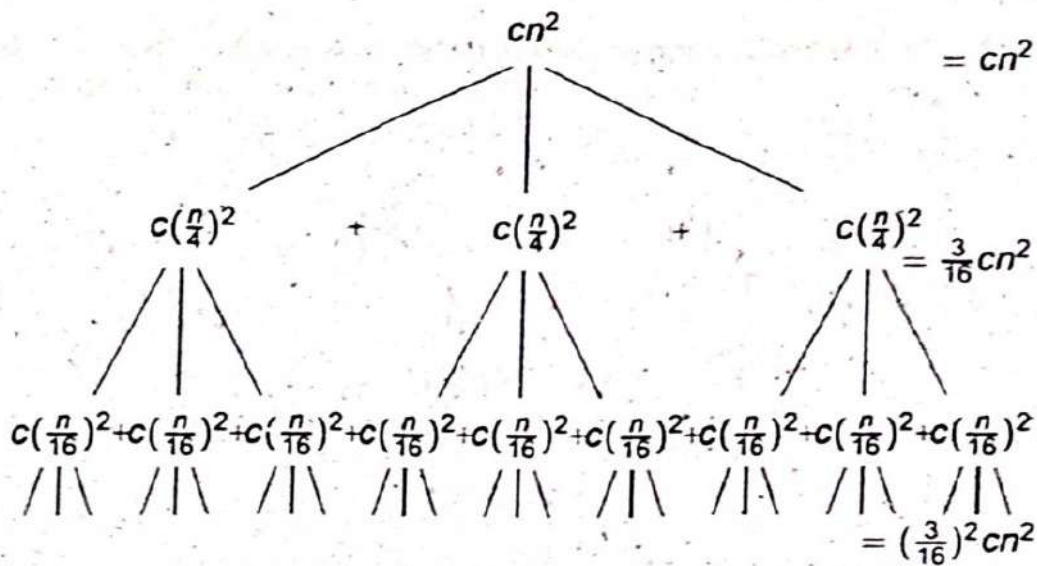
Examples:

1- $3T\left(\frac{n}{4}\right) + n^2$

Expand it into a recursion tree:



Summation of cost at each level:



$$T(n) = cn^2 + \frac{3}{16} cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \dots$$

$$T(n) = cn^2 \left(1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \dots \right)$$

This equation can further be solved by geometric series. For this case $r = \frac{3}{16}$ i.e. $r < 1$, so the formula applied for infinite geometric series would be:

$$S = \frac{a_1}{1 - r}$$

Here $a_1 = 1$, $r = \frac{3}{16}$, then applying geometric series we have:

$$T(n) = \frac{1}{\left(\frac{3}{16}\right)} n^2 = \frac{16}{13} n^2$$

The height of the recurrence tree is $\log_4 n = \log_4 n$. Hence,

$$T(n) = \frac{16}{13} n^2 + \log_4 n$$

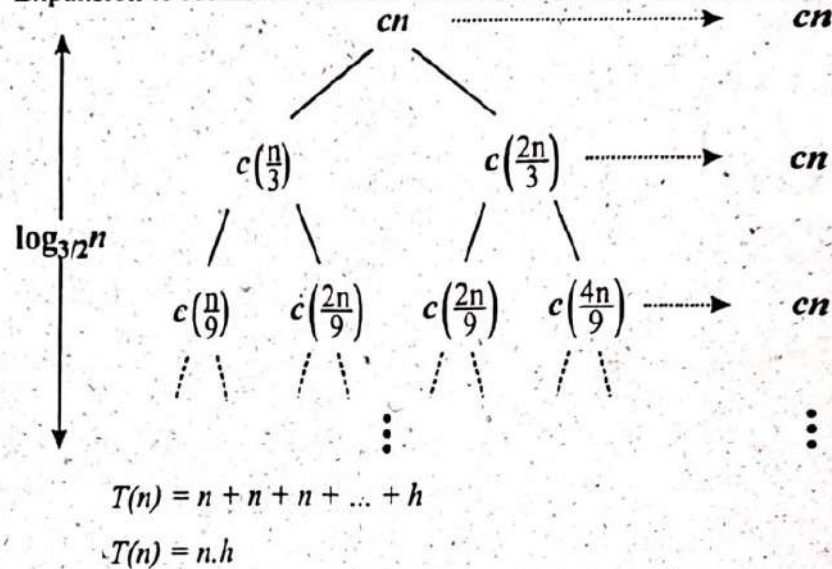
n^2 has the highest growth. Ignore constant, we have

$$T(n) = n^2$$

It can be proved using substitution method with the guess n^2 .

2- $T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$

Expansion to recursion tree and summation of cost at each level is as follow



Here the height of the tree is $\log_{3/2} n$ because $2/3$ is greater than $1/3$. By reversing the greater one is the base of log of height.

$$T(n) = n \log_{3/2} n$$

Note: The substitution method for $T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$ having guess $T(n) = n \log_{3/2} n$ given in the second example of the substitution method.

Short Answers to the Questions

Define algorithm analysis.

A process of analyzing the problem-solving capability of the algorithm in terms of running time (or execution time) and memory space required is called *algorithm analysis*.

What is algorithm complexity?

Complexity of an algorithm is a measurement of the amount of running time and/or space required by an algorithm for an input of a given size (n). An algorithm is said to be efficient and fast if it takes less time to execute and consumes less memory space.

What is meant by log-linear time?

An algorithm is said to run in a *log-linear time* when it calls another *logarithmic algorithm* inside a loop. Log-linear time is also known as *quasi-linear time*.

 What do you know about the order of growth?

The order of growth means how the performance of the algorithm is going to increase or decrease by increasing the input size. For example, $O(1)$ remains constant on increasing or decreasing input size. But $O(n^2)$ increases as the input size increases.

 What is the worst-case?

The maximum (or upper bound) execution time required by an algorithm to complete the task is called *worst-case*. In the worst-case analysis, we calculate the maximum number of steps or operations required by the algorithm on the input data to complete a certain task. For example, in the linear search problem, the worst-case happens when the element to be searched is not present inside the list i.e. an array. Therefore, the algorithm will compare all the elements of the array one by one.

 What is big O-notation?

Big O-notation is a mathematical notation. It describes the limiting behavior of a function when the argument tends towards a particular value or infinity. In computer science, it is used to express the *upper bound* of an algorithm's running time. It is specifically used to measure the *worst-case* time complexity of an algorithm. It describes how the running time of an algorithm grows relative to the input, as the input gets larger.

 What is Step Counts method?

In this method, the running time of each step is calculated and the highest of the running time among all steps is the overall running time of an algorithm. We typically ignore small values, since we are usually interested in estimating how slow the program will be on large inputs.

EXERCISE

Q.1 Select a correct answer from the multiple choices.

1. Space complexity of an algorithm is the maximum amount of _____ required by it during execution.
(a) Time (b) Operations (c) Memory space d) None
2. An algorithm that indicates the amount of temporary storage required for running the algorithm is termed as:
(a) Big Theta θ (f) (b) Space complexity
(c) Big Oh O (f) (d) Time complexity
3. The amount of time the computer needs to run to completion is known as:
(a) Big Theta θ (f) (b) Space complexity
(c) Recursive function (d) Time complexity

4. Two main measures for the efficiency of an algorithm are:
 (a) Processor and memory (b) Complexity and capacity
 (c) Time and space (d) Data and space
5. The time factor when determining the efficiency of algorithm is measured by:
 (a) Counting microseconds (b) Counting the number of key operations
 (c) Counting the number of statements (d) Counting the kilobytes of algorithm
6. Which of the following case does not exist in complexity theory?
 (a) Best-case (b) Worst-case (c) Average-case (d) Null-case
7. The complexity of merge sort is:
 (a) $O(n)$ (b) $O(\log n)$ (c) $O(n^2)$ (d) $O(n \log n)$
8. The complexity of linear search algorithm is:
 (a) $O(n)$ (b) $O(\log n)$ (c) $O(n^2)$ (d) $O(n \log n)$
9. It indicates the log-linear time:
 (a) $O(n)$ (b) $O(\log n)$ (c) $O(n^2)$ (d) $O(n \log n)$
10. It indicates the quadratic time:
 (a) $O(n)$ (b) $O(2^n)$ (c) $O(n^2)$ (d) $O(n^3)$

Answers of MCQs

01 (c)	02 (b)	03 (d)	04 (c)	05 (b)	06 (d)	07 (c)	08 (a)	09 (d)	10
--------	--------	--------	--------	--------	--------	--------	--------	--------	----

- Q.2 Explain time and space complexities with examples.
- Q.3 Differentiate between worst-case and average-case analysis.
- Q.4 Explain commonly used asymptotic notations.

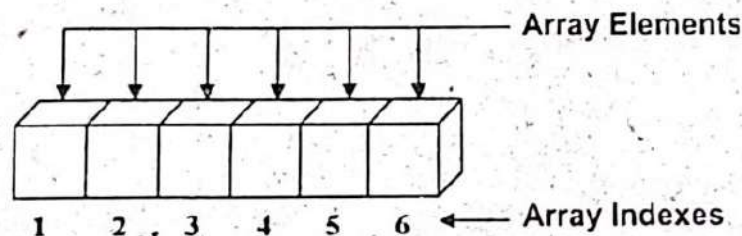
3.1 INTRODUCTION TO ARRAYS

During computer program execution, we store a single value in a single variable. But sometimes we need to store multiple values of the same data type and then process these values in different ways. For example, we may need to store marks of 100 students, and then add up all these marks and finally calculate average marks obtained by the students.

One solution to this sort of problem may be declaring multiple variables of the same data type and using these variables for needed calculations. As you can imagine this solution will be increasingly difficult as the number of involved variables grows higher and higher.

The second solution is using an "array" data structure. It makes the task so simple and easier that any number of values as well as references (addresses of values) of same data type can be handled with a simple logical method. This chapter is all about array data structure, its types, and operations that can be performed on arrays.

An array is the most fundamental abstract data type (ADT). Abstract data type, commonly abbreviated ADT, is a way of classifying data structures based on how it is used and the behaviors it provides. The array has a linear data structure. It is a set of consecutive memory locations with the same name and data type. The memory locations in an array are called *elements* of the array. The total number of elements in the array is called the *size* (or *range*) of the array which remains normally fixed during program execution. Each element of the array is capable of storing only one data item. The simplest form of an array is a one-dimensional array. It is also called a *linear array*.



An array is given a unique name and each element in the array has a unique position number. This position number is called *index* or *subscript*. Each element of the array is accessed by specifying the name of the array and its position number or index. The index must be an unsigned integer value. The general syntax to access an element of a one-dimensional array is described as;

`Array_name[index]`

Most programming languages use index value either 0 or 1 as the first element of an array. In C++, the first element of the array has an index value of 0. Thus in C++, the elements of one-dimensional array 'A' will be denoted as;

$A[0], A[1], A[2], \dots, A[n-1]$

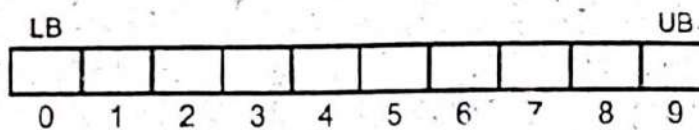
In the above notations, 'A' is the name of the array, and numbers 0, 2, 3, ... n-1 represent indexes of individual elements. The "n-1" represents the index value of the last element of the array.

The index of the first element of the array is called its *lower bound (LB)* and the index of the last element is called its *upper bound (UB)*. The total number of elements in a one-dimensional array can be computed as follows:

$$\text{Total number of elements in an array} = \text{UB} - \text{LB} + 1$$

Suppose an array "abc" of integer data type having 10 elements is declared in C++ as follows:

```
int abc[10];
```



As we can see, index of the first element is 0 and that of the last element is 9. The range or size of the array becomes 10 according to following calculation:

$$\begin{aligned}
 \text{Total number of elements in the array} &= \text{UB} - \text{LB} + 1 \\
 &= 9 - 0 + 1 \\
 &= 10
 \end{aligned}$$



In C++, the lower bound and upper bound cannot be changed during program execution. It means that the size of an array remains fixed at runtime.

3.1.1 Representation of Array in Memory

An array occupies a contiguous block of memory. This memory block is constituted by the memory locations required according to the number of elements in the array. For example, an array of five integer elements will have a memory block of 5 memory locations. Thus a block of memory is divided into equal parts and each part of the block represents one element of the array. In C++, these elements are numbered from 0 to 4.

Each element of the array also has a unique memory address. The address of the first element (or starting address of an array) of an array is called the *base address* of the array. The name of the array is a pointer to this element. Since the elements of an array occupy a continuous block of memory, therefore if the base address of the array is known, the address of any element of the array can be computed. The formula to compute the address of any element of the array is:

$$A[e - 1] = A_0 + \text{size} * (e - 1)$$

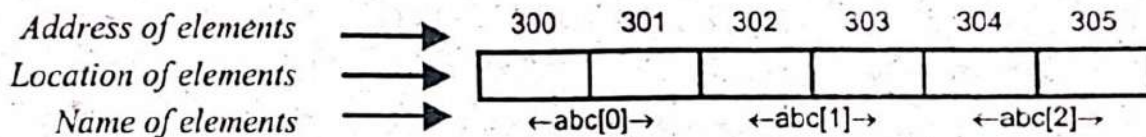
Where

$A[e - 1]$ It represents the name of the element whose we want to find the memory address.

e	It represents the element number of the array whose address we want to find.
A_b	It represents the base address of the array.
size	It represents the size of the array.

Suppose an array "abc" has 3 elements, each of 2 bytes length, then to find out the address of second element of the array 'abc' when the base address is 300 (value of A_b):

$$\begin{aligned}
 \text{abc}[2 - 1] &= 300 + 2 * (2-1) \\
 \text{abc}[1] &= 300 + 2 * 1 \\
 &= 300 + 2 \\
 &= 302
 \end{aligned}$$



We know that each memory location has a unique address. An element of an array may occupy more than one memory locations. It depends upon the data type of the array. In case of "int" type data, each element takes 2 bytes in memory. The address of the element is considered as the address of the first location for that element. In the above array, the known base address of the array "abc" is 300. So the addresses of elements of the array "abc" will be:

- ★ The address of abc[0] is 300
- ★ The address of abc[1] is 302
- ★ The address of abc[2] is 304

3.2 OPERATIONS ON LINEAR ARRAYS

Various operations can be performed on linear arrays (array extending in one direction) for manipulating and managing the involved values. The most important of these operations include traversing, insertion, deletion, searching, and sorting the values.

3.2.1 Traversing Operation

In traversing operation, each element of an array is accessed exactly once for processing. This is also called *visiting* of the array.

An array is traversed to process its data. For example, to compute the sum of values of each element of an array, all elements of the array are accessed exactly once and their values are added. Similarly, to display values of each element of an array, each element of the array is accessed exactly once, and then the values are displayed on the screen.

Arrays are linear data structures, therefore simple steps are required to traverse them. The following algorithm explains the traversing of array.

Suppose there is an array 'abc' with values as shown in the following figure:

2	5	3	8	5	4	5	6	8	5
---	---	---	---	---	---	---	---	---	---

The stepwise addition of elements of the array 'abc' is as follows:

1. Sum = 0
2. Access abc[0], i.e '2' and add to sum=0+2
3. Access abc[1], i.e '5' and add to sum=0+2+5
4. Access abc[2], i.e '3' and add to sum=0+2+5+3
5. Access abc[3], i.e '8' and add to sum=0+2+5+3+8
6. Access abc[4], i.e '5' and add to sum=0+2+5+3+8+5
7. Access abc[5], i.e '4' and add to sum=0+2+5+3+8+5+4
8. Access abc[6], i.e '5' and add to sum=0+2+5+3+8+5+4+5
9. Access abc[7], i.e '6' and add to sum=0+2+5+3+8+5+4+5+6
10. Access abc[8], i.e '8' and add to sum=0+2+5+3+8+5+4+5+6+8
11. Access abc[9], i.e '5' and add to sum=0+2+5+3+8+5+4+5+6+8+5=51

Algorithm – Sum of Linear Array

Write an algorithm that computes the sum of values of elements of a linear array having five elements.

- 1- START
- 2- INPUT Values in array XYZ
- 3- SUM = 0 [Initialize variable SUM to 0]
- 4- REPEAT FOR C = 0 TO 4 [Compute sum of array XYZ]
 SUM = SUM + XYZ[C]
 [End of Loop]
- 5- PRINT SUM [Display the calculated sum]
- 6- END

Program Code 3.1:

```
// This program computes the sum of array
#include <iostream.h>
#include <conio.h>
```

```
class sum
{
private:
    int xyz[5], s;
public:
```



```

sum()
{
    s = 0;
}

// member function to input data into array
input()
{
    cout<<"Enter 5 values & press Enter after typing each value\n";
    for(int i = 0; i<=4; i++)
        cin>>xyz[i];
} //end of input()

// member function to compute and display the sum of array
compute_sum()
{
    for(int i = 0; i<=4; i++)
        s = s + xyz[i];
    cout<<"Sum = "<<s;
} //end of print()

};

void main(void)
{
    sum obj;
    clrscr();
    obj.input(); // Call member function input
    obj.compute_sum(); // Call member function print
    getch();
} //end of main()

```

Output of the program:

Enter 5 values & press Enter after typing each value

5

6

4

3

8

Sum = 26

Algorithm – Even Values of Linear Array

Write an algorithm that displays even values stored in a linear array XYZ having five elements.

- 1- START
- 2- REPEAT Step-3 FOR C = 0 to 4
- 3- IF XYZ[C]%2 = 0 THEN
 PRINT XYZ[C]

```

        END IF
    [End of loop]
4-    END

```

Program Code 3.2:

// This program finds out the even values from the array

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
class even
```

```
{
```

```
private:
```

```
int xyz[5];
```

```
public:
```

```
// member function to input data into array
```

```
input()
```

```
{
```

```
cout<<"Enter 5 values & press Enter after typing each value\n";
for(int i = 0; i<=4; i++)
```

```
cin>>xyz[i];
```

```
}
```

```
// member function to display even numbers
```

```
even_no()
```

```
{
```

```
cout<<"\n Even values are as under:\n";
```

```
for(int i = 0; i<=4; i++)
```

```
if(xyz[i]%2 == 0)
```

```
cout<<xyz[i]<<endl;
```

```
}
```

```
};
```

```
void main(void)
```

```
{
```

```
even obj;
```

```
clrscr();
```

```
obj.input();
```

```
obj.even_no();
```

```
getch();
```

```
} //end of main()
```

Output of the program:

Enter 5 values & press Enter after typing each value

88

23

64

12

53

Even values are as under:

88

64

12

3.2.1.1 Complexity Analysis of Traversal Algorithm of Array

The traversal algorithm of array uses only one loop for traversing linear array with a size of "n". All the elements of an array have to be accessed in the worst-case. Therefore, time complexity and space complexity of array traversal are as follows:

Time Complexity		Space Complexity
Average-Case	Worst-Case	Worst-Case
$\Theta(n)$	$O(n)$	$O(n)$

3.2.2 Insertion Operation

In insertion operation, new data items are added or inserted into any location of an empty or filled array (An array in which data is already entered is called a *filled array*). In case of an empty array, new values are stored in empty memory locations. In case of a filled array, old values are either replaced by new values or pushed forward or backward to make room for the new values. It is very easy to insert data at the end of an array without disturbing the data of other elements of the array.

3.2.2.1 Inserting Data Item at the End of Array

Consider an array "abc" having 10 elements with its first three elements having values as shown below:

Value	16	77	66							
Index	0	1	2	3	4	5	6	7	8	9

To insert a value 81 at the end of the array, i.e. at location "abc[3]", the assignment statement is written as:

abc[3] = 81;

The new values can easily be inserted in the above array from fourth to tenth elements as these elements are empty.

Algorithm – Insertion at End

Write an algorithm to add 7 values into elements at the end of array 'ABC' that has only three" items stored in its first three elements.

- 1- START
- 2- REPEAT Step-3 TO 4 FOR I = 3 to 9
- 3- INPUT value in N
- 4- ABC[I] = N
[End of Step-2 Loop]
- 5- END

Program Code 3.3:

```
#include <iostream.h>
#include <conio.h>

class temp
{
    private:
        int abc[10];
    public:
        assign()
        {
            abc[0] = 66;
            abc[1] = 72;
            abc[2] = 36;
        }

        // member function to insert values,
        insert()
        {
            cout<<"Enter 7 values & press Enter after typing each value\n";
            for(int i = 3; i<=9; i++)
                cin>>abc[i];
        } //end of input()

        // member function to print values
        display()
        {
            for(int i = 0; i<=9; i++)
                cout<<abc[i]<<endl;
        } //end of display()
};

void main(void)
```



```

{
    temp obj;
    clrscr();
    obj.assign();
    obj.insert();
    cout<<"Values in Array "<<endl;
    obj.display();
    getch();
} //end of main()

```

3.2.2.2 Inserting Value at a Specified Location in an Array

A new value can be inserted at a specified location in an array. To insert a new value, the values of all the existing elements may be moved one step forward or backward to make room (or space) for the new value. The following algorithm explains this process.

Algorithm - Inserting a Value at a Specified Location

Write an algorithm that inserts a value at a specified location in an array XYZ having 'N' elements.

- 1- START
- 2- INPUT location into P
- 3- [Check the value in location P whether it is a valid position in an array 'XYZ']
 - IF (P >= N) THEN
 - PRINT "Invalid position"
 - Return to Step-2
 - END IF
- 4- INPUT value in VAL
- 5- [Move values of elements from P one step forward through loop]
 - REPEAT WHILE (N >= P)
 - XYZ[N+1] = XYZ[N]
 - N = N+1
 - END WHILE
- 6- [Insert value N at position P using assignment statement]
 - XYZ[P] = VAL
- 7- END

Program Code 3.4:

```

#include <iostream.h>
#include <conio.h>

class temp
{

```

```
private:
    int xyz[5];
public:

    // member function to assign values to array
    assign(int p[])
    {
        for(int i = 0; i <= 3; i++)
            xyz[i] = p[i];
    } //end of assign()

    // member function to insert a value into array
    insert(int loc, int val)
    {
        for(int i = 4; i >= loc; i--)
            xyz[i+1] = xyz[i];
        xyz[loc] = val;
    } //end of insert()

    // member function to print all values
    display(int n)
    {
        for(int i = 0; i <= n; i++)
            cout << xyz[i] << endl;
    } //end of display()

};

void main(void)
{
    temp obj;
    int pos, n, a[4] = {44, 55, 6, 5};
    clrscr();
    obj.assign(a);
    cout << "Values before insertion" << endl;
    obj.display(3);
    cout << "Enter value to insert? ";
    cin >> n;
    cout << "Enter position to insert? ";
    cin >> pos;
    if(pos >= 5)
    {
        cout << "Invalid Location "; return 0;
    }
    obj.insert(pos, n);
}
```



```

    cout<<"Values after insertion"<<endl;
    obj.display(4);
    getch();
} //end of main()

```

3.2.2.3 Complexity Analysis of Insertion Algorithm of Array

The insertion algorithm of array uses only one loop for inserting a new value into a specific location of a linear array with size 'n'. Suppose we want to insert a new value in the first element of the filled array, then values of 'n' elements in the array must be shifted one place to the right before the new value can be added. Similarly, if we want to insert a new value at the end of the array, then all the values of the array must be traversed first to reach the required element. The time complexity and space complexity of this algorithm are as follows:

Time Complexity		Space Complexity
Average-Case	Worst-Case	Worst-Case
$\Theta(n)$	$O(n)$	$O(n)$

3.2.3 Deletion Operation

In deletion operation, the data item from a particular element of an array is removed. Like insertion, deletion can be performed at the end or any position in the array.

3.2.3.1 Deleting Items at the End

Deleting or removing a data item at the end of the array is simple and easy. To remove the data item from the last position of the array, the last element is accessed and deleted by placing a null value.

Consider an array of names of 5 countries as shown below:

Russia	England	America	Pakistan	India
0	1	2	3	4

To remove the data item from the last position of the array (i.e. India), move to the last element and then delete the data item by placing the null value. After deletion of the last value the array becomes as under:

Russia	England	America	Pakistan	
0	1	2	3	4

3.2.3.2 Deleting Item from a Specified Location

When a data item is removed from a specified location within an array, the items from that location up to the end of the array are moved one location towards the beginning of the array.

Consider an array of names of 5 countries as shown below:

Russia	England	America	Pakistan	India
0	1	2	3	4

To delete data item "America" at position 2, the contents of locations 3 and 4 are moved one step towards the beginning of the array, i.e. "Pakistan" will be shifted to location 2 and "India" to location 3. Delete data item from the last position of the array by placing the null value. Thus the above array becomes as under:

Russia	England	Pakistan	India	
0	1	2	3	4

Algorithm - Deleting an Item from Array

Write an algorithm that deletes the data item at a specified location of an array having N elements.

1. INPUT location in LOC
2. IF LOC > N THEN
 PRINT "Invalid location"
 RETURN
END IF
3. [Now move each element from the specified location one step towards the beginning.]
 REPEAT FOR C = LOC TO N-1
 COUNTRY[C] = COUNTRY[C+1]
 [End of Loop]
 COUNTRY[N-1] = NULL
4. EXIT

Program Code 3.5:

```
// Program that deletes a value from a specified location of an array
#include <iostream.h>
#include <conio.h>

class abc
{
    private:
        int arr[5];
    public:
        // member function to assign values
        assign(int p[])
        {
            for(int i = 0; i<=4; i++)
```



```

        arr[i] = p[i];
    } //end of assign()

    // member function to delete a value
    del(int loc)
    {
        for(int i = loc; i<=4; i++)
            arr[i] = arr[i+1];
        arr[4] = 0;
    } //end of del()

    // member function to print values of array
    display()
    {
        for(int i = 0; i<=4; i++)
            cout<<arr[i]<<endl;
    } //end of display()
};

void main(void)
{
    abc obj;
    int pos, n, a[5] = {44, 55, 6, 3, 66};
    clrscr();
    obj.assign(a);
    cout<<"Values before deletion"<<endl;
    obj.display();
    cout<<"Enter position to delete? ";
    cin>>pos;
    if(pos >=5)
    {
        cout<<"Invalid Location ";
        return 0;
    }
    obj.del(pos-1);
    cout<<"Values after deletion"<<endl;
    obj.display();
    getch();
} //end of main()

```

3.2.3.3 Complexity Analysis of Deletion Algorithm of Array

The deletion algorithm of array uses only one loop for deleting the value from a linear array with size "n". Suppose we want to delete a value from the first element, then values of 'n-1' array elements must be shifted one place left. The time complexity and space complexity of this algorithm are as follows:

Time Complexity		Space Complexity
Average-Case	Worst-Case	Worst-Case
$\Theta(n)$	$O(n)$	$O(n)$

3.2.4 Search Operation

Searching values is a very important operation that is needed frequently in any kind of software development. The process of finding a specific data item and its location in an array is called *searching*. The search is "successful" if the specified data item is found. The search operation is terminated as soon as the data item is found, however, duplicate values may also exist in the array. Here in our examples, we assume that no duplicate values exist in the array. If the specified data item is not found, the search is declared as "unsuccessful."

Usually, the search operation in data structures is used for data modification along with insertion, deletion, and update processes. For example, when a data item is to be deleted, it is first searched and then deleted, if found.

A variety of search algorithms can be used by software developers. The most commonly used search algorithms are as follows:

- i. Sequential Search
- ii. Binary Search

3.2.5 Sorting Operation

The process of arranging data items of an array in a specific order is called *sorting*. The data items in an array are usually sorted into ascending or descending order.



The details on searching and sorting are given in the chapter "Searching and Sorting".

3.3 MULTI-DIMENSIONAL ARRAYS

An array can have multiple subscripts to represent more than one dimension. An array having more than one dimension is called a *multi-dimensional array*. Dimensions show layers of arrays. For example, a table may be considered as a two-layered array (i.e. two-dimensional array). The simplest form of the multi-dimensional array is the two-dimensional array. In the real world, a calendar of a particular month is a two-dimensional array where numbers along rows represent days and columns represent weeks of the month.

2013						
January						
S	M	T	W	T	F	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

A two-dimensional array is an array of a one-dimensional array. It consists of rows and columns. A two-dimensional array is called *matrix* in mathematics and *table* in business applications. Hence, two-dimensional arrays are also called a *matrix* or *table*.

In C++, the element of a two-dimensional array is referenced by two index values. One index value represents the row and the second represents the column. A matrix with the same number of rows and columns is called a *square matrix*.

A two-dimensional array "X" having 3 rows and 2 columns is shown below.

X[0][0]	X[0][1]
X[1][0]	X[1][1]
X[2][0]	X[2][1]

In general, an array with "m" rows and "n" columns is called an *m-by-n array*. The total number of elements of a two-dimensional array having "m" rows and "n" columns is $n \times m$. For example, an array having 3 rows and 6 columns has $3 \times 6 = 18$ elements.

The total number of bytes occupied by a table in memory is computed as;

$$\text{bytes} = \text{number of rows} * \text{number of columns} * \text{size of data type}$$

For example, if a table has 6 rows, 4 columns and the data type is double (i.e., data type size is 8) then the size of the table is:

$$\text{bytes} = 6 \times 4 \times 8 = 24 \times 8 = 192$$

In C++, a two-dimensional array is declared by specifying two indexed values, each enclosed in square brackets. The first indexed value identifies the total number of rows and the second identifies the total number of columns. For example, to declare a double data type two-dimensional array "temp" having 6 rows and 4 columns, the declaration statement is written as;

```
double temp[6][4];
```

The table "temp" with rows and columns and subscripted values is given below.

temp[0][0]	temp[0][1]	temp[0][2]	temp[0][3]
temp[1][0]	temp[1][1]	temp[1][2]	temp[1][3]
temp[2][0]	temp[2][1]	temp[2][2]	temp[2][3]
temp[3][0]	temp[3][1]	temp[3][2]	temp[3][3]
temp[4][0]	temp[4][1]	temp[4][2]	temp[4][3]
temp[5][0]	temp[5][1]	temp[5][2]	temp[5][3]

The total number of elements of table "temp" are: $6 \times 4 = 24$.

Program Code 3.6:

Write a program that inputs data into a table and displays on the screen in tabular form.

```
#include <iostream.h>
#include <conio.h>
```

```

void main(void)
{
    int table[3][4];
    int r, c;
    clrscr();
    r = 0;
    // input values in table
    while(r<=2)
    {
        for(c = 0; c<= 3; c++)
        {
            cout<<"Enter value for row "<<r<<" and column "<<c<<" ? ";
            cin>>table[r][c];
        } //end of for loop
        r++;
    } //end of while loop

    // display values from table
    cout<<"\nValues in tabular form" <<endl;
    for(r = 0; r<= 2; r++)
    {
        for(c = 0; c<=3; c++)
            cout<<table[r][c]<<"\t";
        cout<<endl;
    }
} //end of main()

```

Output of the program:

```

Enter value for row 0 and column 0 ? 36
Enter value for row 0 and column 1 ? 45
Enter value for row 0 and column 2 ? 25
Enter value for row 0 and column 3 ? 32
Enter value for row 1 and column 0 ? 78
Enter value for row 1 and column 1 ? 25
Enter value for row 1 and column 2 ? 96
Enter value for row 1 and column 3 ? 45
Enter value for row 2 and column 0 ? 78
Enter value for row 2 and column 1 ? 65
Enter value for row 2 and column 2 ? 12
Enter value for row 2 and column 3 ? 45

```

Values in tabular form

36	45	25	32
78	25	96	45
78	65	12	45

Program Code 3.7:

Write a program that inputs data into a table and finds out the largest and the smallest values entered in the table.


```

#include <iostream.h>
#include <conio.h>
void main(void)
{
    int table[3][4];
    int r, c, max, min, mx_r, mx_c, mn_r, mn_c;
    clrscr();
    r = 0;

    // input values in a table
    while(r<=2)
    {
        for(c = 0; c<=3; c++)
        {
            cout<<"Enter value for row "<<r<<" and column "<<c<<" ? ";
            cin>>table[r][c];
        } //end of for loop
        r++;
    } //end of while loop

    // find maximum & minimum values
    max = min = table[0][0];
    for(r = 0; r<=2; r++)
        for(c = 0; c<=3; c++)
        {
            if(max<table[r][c])
            {
                max = table[r][c];
                mx_r = r;
                mx_c = c;
            }
            if(min>table[r][c])
            {
                min = table[r][c];
                mn_r = r;
                mn_c = c;
            }
        }

    cout<<"\nThe Largest value "<<max<<endl;
    cout<<"Location["<<mx_r<<"]["<<mx_c<<"]\n";
    cout<<"\nThe Smallest value "<<min<<endl;
    cout<<"Location["<<mn_r<<"]["<<mn_c<<"]\n";
} //end of main()

```

Output of the program:

```

Enter value for row 0 and column 0 ? 52
Enter value for row 0 and column 1 ? 62
Enter value for row 0 and column 2 ? 58
Enter value for row 0 and column 3 ? 45
Enter value for row 1 and column 0 ? 62
Enter value for row 1 and column 1 ? 85
Enter value for row 1 and column 2 ? 14

```

Enter value for row 1 and column 3 ? 52
 Enter value for row 2 and column 0 ? 63
 Enter value for row 2 and column 1 ? 78
 Enter value for row 2 and column 2 ? 95
 Enter value for row 2 and column 3 ? 158

The Largest value 158

Location [2][3]

The Smallest value 14

Location [1][2]

3.3.1 Complexity Analysis of 2-D Array

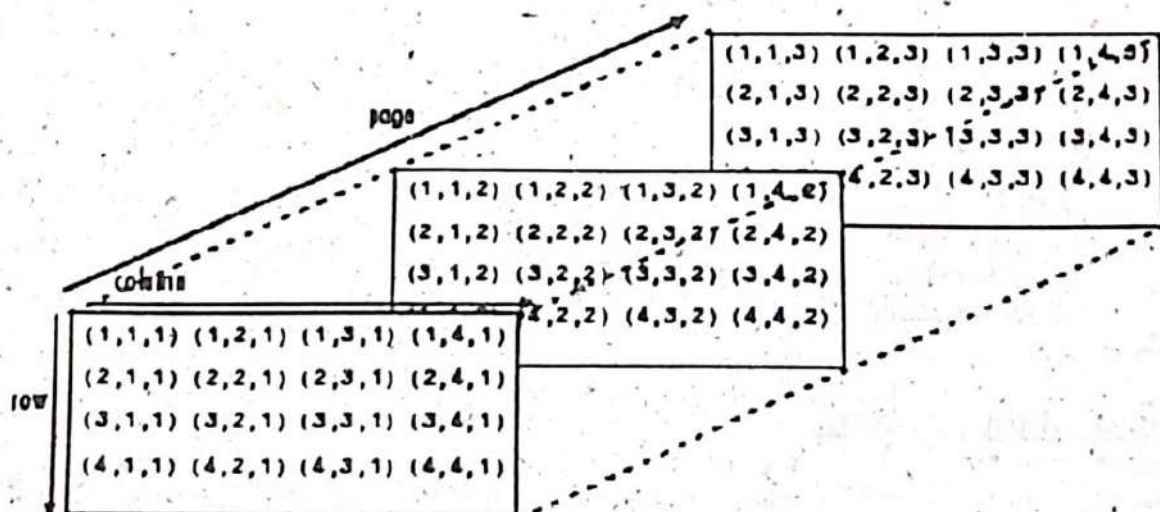
The time complexity and space complexity for traversing, insertion, and deletion operations of the two-dimensional array with size $n \times m$ (if $m = n$, then $n \times n = n^2$) are as follows:

Time Complexity		Space Complexity
Average-Case	Worst-Case	Worst-Case
$\Theta(n^2)$	$O(n^2)$	$O(n^2)$

3.3.2 Higher Dimensional Arrays

Arrays can have more than two subscripted values. A three-dimensional array is an array of two-dimensional arrays (tables). It requires three subscripts to represent its elements.

- ★ The first references dimension 1, the row.
- ★ The second references dimension 2, the column.
- ★ The third references dimension 3. This illustration uses the concept of a *page* to represent dimensions 3 and higher.



To access the element in the second row, the third column of page 2, for example, we use the subscripts [2][3][2].

In the real world, a calendar of a particular year is an example of a three-dimensional array. Each page of the calendar, 1 through 12, is an element, representing a month, which contains approximately 30 elements, which represent days. Each day may or may not have information in it. So one subscript is required to represent days, second subscript for weeks, and third subscript for months.

To declare an integer data type three-dimensional array "calendar13" having 60 rows (weeks), 7 columns (days), and 12 tables (months), the declaration statement is written as:

```
int calendar13[60][7][12];
```

Other operations on three or higher dimensional arrays are same as on two-dimensional arrays.

2013																				
January							February							March						
S	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S
1	2	3	4	5			1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	7	8	9	10	11	12	8	9	10	11	12	13	14	15	16	17	18	19	20	21
13	14	15	16	17	18	19	15	16	17	18	19	20	21	22	23	24	25	26	27	28
20	21	22	23	24	25	26	22	23	24	25	26	27	28	29	30	31				
27	28	29	30	31			29	30	31											
April							May							June						
S	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S
1	2	3	4	5	6		1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	8	9	10	11	12	13	8	9	10	11	12	13	14	15	16	17	18	19	20	21
14	15	16	17	18	19	20	15	16	17	18	19	20	21	22	23	24	25	26	27	28
21	22	23	24	25	26	27	22	23	24	25	26	27	28	29	30	31				
28	29	30					29	30	31											
July							August							September						
S	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S
1	2	3	4	5	6		1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	8	9	10	11	12	13	8	9	10	11	12	13	14	15	16	17	18	19	20	21
14	15	16	17	18	19	20	15	16	17	18	19	20	21	22	23	24	25	26	27	28
21	22	23	24	25	26	27	22	23	24	25	26	27	28	29	30	31				
28	29	30	31				29	30	31											
October							November							December						
S	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S
1	2	3	4	5	6		1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	8	9	10	11	12	13	8	9	10	11	12	13	14	15	16	17	18	19	20	21
14	15	16	17	18	19	20	15	16	17	18	19	20	21	22	23	24	25	26	27	28
21	22	23	24	25	26	27	22	23	24	25	26	27	28	29	30	31				
28	29	30	31				29	30	31											

3.3.3 Algebraic Operations on Matrices

The algebraic operations like addition, subtraction multiplication, etc. can be easily performed on matrices. The matrices must obey certain rules for performing these operations. These rules are as follows:

- ★ To add/subtract two matrices, the number of rows & columns of the first matrix must be equal to the number of rows & columns of the second matrix.
- ★ To multiply two matrices, the number of columns of the first matrix must be equal to the number of rows of the second matrix.

For example, to add two matrices A and B, the corresponding elements of these matrices are added:

$$A = \begin{bmatrix} 1 & 6 \\ 2 & 3 \end{bmatrix}, \quad B = \begin{bmatrix} 2 & 5 \\ 1 & 2 \end{bmatrix}$$

$$A + B = \begin{bmatrix} 1+2 & 6+5 \\ 2+1 & 3+2 \end{bmatrix}$$

$$A + B = \begin{bmatrix} 3 & 11 \\ 3 & 5 \end{bmatrix}$$

Program Code 3.8:

Write a program that initializes values in two tables A and B and adds tables A and B and store result into table C.

```
#include <iostream.h>
#include <conio.h>
void main(void)
```

```

{
    int A[3][3] = {{2, 3, 8}, {5, 8, 3}, {2, 6, 3}};
    int B[3][3] = {{2, 3, 1}, {6, 1, 2}, {6, 2, 1}};
    int C[3][3];
    clrscr();
    void addition(int[3][3], int[3][3], int [3][3]);
    void print(char [15], int [3][3]);
    addition(A,B,C);
    print("Matrix A: ", A);
    print("Matrix B: ", B);
    print("addition of A & B ", C);
} // end of main()

// definition of addition() function
void addition(int X[3][3], int Y[3][3], int Z[3][3])
{
    int r, c;
    for(r = 0; r<=2; r++)
        for(c = 0; c<=2; c++)
            Z[r][c] = X[r][c] + Y[r][c];
} // end of addition()

// definition of print() function
void print( char str[], int R[3][3])
{
    int r, c;
    cout<<endl<<str<<endl;
    for(r = 0; r<=2; r++)
    {
        for(c = 0; c<=2; c++)
            cout<<R[r][c]<<"\t";
        cout<<endl;
    }
} // end of print()

```

Output of the program:

Matrix A:

2	3	8
5	8	3
2	6	3

Matrix B:

2	3	1
6	1	2
6	2	1

Addition of A & B

4	6	9
11	9	5
8	8	4

4 LIMITATIONS OF ARRAYS

We can use arrays whenever we have to store and manipulate collections of elements. However, the use of arrays for this purpose presents some limitations also. Some limitations of arrays are as follows:

- ★ The dimension of an array is determined at the moment the array is created, and cannot be changed during program execution.
- ★ The array occupies an amount of memory that is proportional to its size, independently of the number of elements that are needed.
- ★ If we want to keep the elements of the collection in order, and insert a new value in its correct position, or remove it, then, for each such operation we may need to move many elements forward or backward.

5 APPLICATIONS OF ARRAYS

Following are some applications of arrays:

- Arrays are used to implement mathematical vectors and matrices, as well as other kinds of rectangular tables. Many small and large databases consist of one-dimensional arrays whose elements are called records.
- Arrays are used to implement other data structures, such as lists, heaps, hash tables, deques, queues, stacks, and strings. Array-based implementations of other data structures are very simple and space-efficient requiring little space but may have poor space complexity.
- Arrays can be used to determine partial or complete control flow in programs. They are known in this context as control tables.

Summary --- Complexities of Linear Array

Linear Array

Operation	Time Complexity		Space Complexity
	Average-Case	Worst-Case	Worst-Case
Access	$\Theta(1)$	$O(1)$	$O(n)$
Traversal	$\Theta(n)$	$O(n)$	$O(n)$
Search	$\Theta(n)$	$O(n)$	$O(n)$
Insertion	$\Theta(n)$	$O(n)$	$O(n)$
Deletion	$\Theta(n)$	$O(n)$	$O(n)$

Short Answers to the Questions



What is an array?

An array is a linear data structure. It is a set of consecutive memory locations with the same name and data type. The memory locations in an array are called *elements* of the array. The total number of elements in the array is called the *size* or *range* of the array which remains normally fixed during program execution.



Why an array is traversed?

An array is traversed to process its data. For example, to compute the sum of values of each element of an array, all elements of the array are accessed exactly once and their values are added. Similarly, to display values of each element of an array, each element of the array is accessed exactly once and the values are displayed on the screen.



What is a multi-dimensional array?

An array having more than two dimensions is called a *multi-dimensional array*. Dimensions show layers of arrays. For example, a table may be considered as a two-layered array (i.e. two-dimensional array). The simplest form of the multi-dimensional array is the two-dimensional array.



Describe some limitations of arrays?

Some limitations of arrays are as follows:

- ★ The dimension of an array is determined the moment the array is created, and cannot be changed during program execution.
- ★ The array occupies an amount of memory that is proportional to its size independently of the number of elements that are needed.
- ★ If we want to keep the elements of the collection in order, and insert a new value in its correct position, or remove it, then, for each such operation we may need to move many elements forward or backward.

EXERCISE

Q.1 Select the correct answer from the multiple choices.

1. Which one is true about an array?

- | | |
|-----------------------------------|---------------------------------------|
| (a) It is a linear data structure | (b) It is a non-linear data structure |
| (c) Size of an array is unlimited | (d) None of these |

In C++, the first element of an array has an index value of:

- | | |
|--------|-------------------|
| (a) 1 | (b) 0 |
| (c) -1 | (d) None of these |

3. The index of the first element of an array is called:
 - (a) Upper Bound
 - (b) Lower Bound
 - (c) Subscript
 - (d) None of these
4. The address of the first element of an array is called:
 - (a) Virtual address
 - (b) Standard address
 - (c) Initial address
 - (d) Base address
5. An array in which data is already entered is called:
 - (a) Full array
 - (b) Empty array
 - (c) Filled array
 - (d) None of these
6. How many index values are required to access a specific element of a table?
 - (a) 1
 - (b) 2
 - (c) 3
 - (d) 4
7. Two-dimensional array is also called:
 - (a) table
 - (b) matrix
 - (c) vector
 - (d) both (a) & (b)
8. A matrix with the same number of rows and columns is called:
 - (a) Complete matrix
 - (b) Full matrix
 - (c) Zero matrix
 - (d) None of these
9. Finding a specific data item and its location in an array is called:
 - (a) Searching
 - (b) Traversing
 - (c) Visiting
 - (d) None of these
10. Accessing each element of an array exactly once is called:
 - (a) Searching
 - (b) Traversing
 - (c) Visiting
 - (d) both (b) & (c)
11. The memory address of the first element of an array is called:
 - (a) floor address
 - (b) foundation address
 - (c) first address
 - (d) base address

Answers of MCQs

01 (a)	02 (b)	03 (b)	04 (d)	05 (c)	06 (b)	07 (d)	08 (d)	09 (a)	10 (d)
11 (d)									

Q.2 What is an array? How is a linear array represented in the computer memory?

Q.3 Write an algorithm and program in C++ to traverse an array in reverse order.

- Q.4 Explain with example the insertion operation in linear arrays.
- Q.5 Explain with example the deletion operation in linear arrays.
- Q.6 Write a C++ program that inputs values into a 2-dimensional array and displays these values on the screen in tabular form.
- Q.7 Write an algorithm that calculates the total number of odd and even values in an array.
- Q.8 Write an algorithm that calculates the factorial of each element of integer array of N elements.
- Q.9 Write an algorithm that traverses an integer array with N elements and finds whether or not the accessed value is a prime number.
- Q.10 Write a C++ program that merges two one-dimensional-arrays of five elements each into one array of 10 elements.
- Q.11 Discuss the advantages and limitations of arrays in detail.
- Q.12 Discuss applications of arrays.

4.1 STRINGS

A string is a sequence of characters. It may include alphabetic letters (a-z), numeric digits (0-9) and various special characters such as +, -, *, ^, /, #, \$ etc. A string constant like *"have a nice day"* is written enclosed in double quotation marks. A string variable is declared to store a string in memory.

Strings have many characteristics associated with them which help to identify and perform different operations on strings. For example, the total number of characters in a string is called *string length*. The length of the string "Pakistan" is eight characters and the length of the string "India" is five characters. A string with no character in it has zero length and is called an *empty* or *null string*.

In C++, the end of a string is always indicated by a special character which is called a *null character*. The null character is denoted by '\0' (backslash and zero). Thus when data is stored into a string variable, the null character at the end of the string will be appended automatically. Thus if a string variable is declared with 10 characters, it can store only 9 characters; the tenth character will be '\0'.

The string is an important data structure in various applications like word processing, graphics and web applications, etc. We need to perform different operations on strings in almost all the applications. These operations include concatenation (joining) of strings, finding a substring in a string, deleting leading or trailing characters in a string, etc. Therefore, efficient representation and handling of strings is an important topic in data structures.

4.1.1 Representation of Strings in Memory

In data structures, a string is implemented as an array of characters. It is represented in memory as a sequence of storage locations of a character data type. Each storage location stores one character of the string i.e. each character taking one byte of memory.

For example, the string "DATA STRUCTURE" in memory is represented as;

D	A	T	A		S	T	R	U	C	T	U	R	E	'\0'	
---	---	---	---	--	---	---	---	---	---	---	---	---	---	------	--

Like other arrays, the individual characters of strings can easily be processed or manipulated with the help of the position of characters in the string or index value in the string/array of characters.

4.1.2 Types of Strings

Different types of strings are as follows:

- i) Fixed-length string
- ii) Variable-length string
- iii) Linked string

i) Fixed-Length String

The string which has a fixed length and uses the fixed amount of memory (whether this maximum is reached or not) is called a *fixed-length string*.

In fixed-length string, the amount of memory required for storing the string is fixed at the time the string variable is declared. Its length remains fixed throughout the execution of the program. For example, to declare a string variable "strvar" of size 15 characters and to store string "PAKISTAN" in it, C++ statement is written as:

```
char strvar[15] = "PAKISTAN";
```

In memory, this string is represented as:

P	A	K	I	S	T	A	N	\0							
---	---	---	---	---	---	---	---	----	--	--	--	--	--	--	--

As we can see, nine locations are occupied by the string in the memory, eight locations for "PAKISTAN" and one location for the null character. Six locations remain empty.

This way of storing and processing string usually results in wastage of a large amount of memory. It also creates problems while processing strings and therefore, slows down the accessing speed.

ii) Variable-Length String

The string whose length is not explicitly fixed and uses a varying amount of memory depending on its actual size is called a *variable-length string*.

A variable-length string is a more dynamic and powerful type of string. The string variable of this type is declared without specifying the string size. Most modern programming languages use variable-length strings.

The storage of variable-length string in memory is handled in different ways. Most commonly used methods are as follows:

- ★ Boundary Marker
- ★ String Descriptor

A boundary marker is a character. It is used to specify the end of the string. Usually, the character ↑ (or \$\$ double dollar signs) is used as a boundary marker. In this method, strings are stored in memory one after the other separated by boundary markers ↑ (or \$).

Using the Boundary Marker method, two strings "PAKISTAN" and "INDIA" are stored in the memory as:

P	A	K	I	S	T	A	N	\0	↑	I	N	D	I	A	\0
---	---	---	---	---	---	---	---	----	---	---	---	---	---	---	----

On the other hand, in the string descriptor method, "a pointer array" is used to represent the strings having variable lengths. In this method, two fields are used to represent a single string:

Length Field It contains the length of the string.

Pointer Field It contains the memory address of the string i.e. address of the first character of the string. The pointer field points to the memory area where the actual information of the string is stored. The string is accessed through the pointer field.

Following figure represents the string "I LOVE PAKISTAN" by using the string descriptor method.

Length	Pointer	→	I		L	O	V	E		P	A	K	I	S	T	A	N
15	0																

iii) Linked Strings

In linked storage of strings, a string is stored in a sequence of memory cells called *nodes*. Each node stores a fixed number of characters of the string. Each node also contains an element called the *link*. The link points to the next node that contains the next part of the string.

Following diagram shows the storage of string "ABCD MNOP" in linked storage with four characters per node:



Note: Linked storage technique is discussed in the chapter "Linked List".

4.2 STRING OPERATIONS

Although a string is treated as an array, however, the string operations are different from those of the arrays that store numeric data. In arrays of numeric data type, individual elements are accessed and processed, whereas in strings, mostly a group of consecutive elements, called substrings, are accessed for processing.

Some important operations that can be applied on strings are as follows:

- ★ Computing length of string
- ★ Copying a string into another
- ★ Concatenating strings
- ★ Extracting a substring from a given string
- ★ Pattern matching
- ★ Inserting a string into another string

- ★ Deleting a substring from a given string
- ★ Searching and replacing a substring in a given string

4.2.1 Computing Length of a String

The total number of characters or symbols in a string is called *string length*. Computing the length of a string is an important string operation. It is usually performed before applying several other string operations. For example, to insert characters or symbols in a string, the string length is first computed and then the insertion operation is performed.

To compute the length of the string, it is traversed and its characters are counted starting from the beginning and going up to the end of the string, i.e. up to the null character.

Algorithm - Computing Length of a String

Write an algorithm that computes the length of a string entered by a user during program execution.

- 1- START
- 2- INPUT string in variable STR
- 3- L = 0
- 4- REPEAT Step-5 WHILE STR[L] != NULL
- 5- L = L + 1
- [End of Step-4 Loop]
- 6- PRINT "Length of string = ", L
- 7- END

Program Code 4.1

```
// Program that computes the length of a string
#include <iostream.h>
#include <conio.h>
class strlen
{
    public:

        // member function for computing length of string
        int len(char str[])
        {
            for(int c = 0; str[c] != '\0'; c++);
            return c;
        }
};

void main(void)
{
    strlen obj;
```

Program's Output

Enter string : Nabeela
Length of string is = 7


```

char s[80];
clrscr();
cout<<"Enter string : ";
cin>>s;
cout<<"Length of string is = "<<obj.len(s);
getch();
} //end of main()

```

4.2.2 Copying Strings

The process of copying the contents of one string to another is called *string copying*. The string whose contents are copied is called the *source string* whereas the string into which the contents are copied is called the *destination string* or *target string*. The entire source string or part of the source string can be copied to the target string as well.

Algorithm - Copying String

Write an algorithm to copy the contents of one string to another empty string.

- 1- START
- 2- INPUT string in variable STR1
- 3- C = 0
- 4- REPEAT Steps-5 to 7 WHILE STR1[C] != NULL
- 5- STR2[C] = STR1[C]
- 6- C = C + 1
- 7- STR2[C] = NULL
- [End of Step-4 Loop]
- 8- PRINT "Copied string"; STR2
- 9- END

Program Code 4.2

```

// Program that copies one string to another
#include <iostream.h>
#include <conio.h>

class strcopy
{
    public:

        // member function for copying string
        cpyst( char str1[], char str2[])
        {
            for(int i = 0; str1[i]!='\0'; i++)

```

```

        str2[i] = str1[i];
        str2[i] = '\0';
    }
};

void main(void)
{
    strcpy obj;
    char s1[80], s2[80];
    clrscr();
    cout<<"Enter first string : ";
    cin>>s1 ;
    obj.cpystr(s1,s2);
    cout<<"Copied string is:"<<s2;
    getch();
} //end of main()

```

Program's Output

Enter first string : Pakistan
Copied string is: Pakistan

4.2.3 String Concatenation

The process of combining the contents of two or more strings into a single string is called *string concatenation*. It is an important string operation. In real-world applications, for example, we usually store First Name, Middle Name, and Last Name separately in three different string variables or database fields. In order to display Full Name, we shall need to join or concatenate these string values.

The operator 'O' or '||' is used to denote the concatenation of two strings. For example, if S1 = "ABC" and S2 = "XYZ" then;

S1 || S2 = ABCXYZ

The strings are joined without blank space between them. If two concatenated strings are to be separated by a space, a blank space character must be inserted as shown below:

S1 || " " || S2 = ABC XYZ

The symbol " " represents the blank space character. The space character is inserted at the end of the first string.

Similarly, more than two strings can also be concatenated into a single string. Suppose, S1 = "C++", S2 = "Visual Basic 6.0", and S3 = "& Web Programming", then to concatenate these strings and to put the result into a string variable STR, we use:

STR = S1 || " " || S2 || " " || S3

The result stored in string variable STR will be "C++, Visual Basic 6.0 & Web Programming". In this example, more than one "||" operators have been used to combine the strings S1, S2 and S3. The blank space characters are inserted at the end of strings S1 & S2.

Algorithm - Concatenation

Write an algorithm to combine three strings into a single string.

- 1- START
- 2- INPUT three strings in variables S1, S2 and S3
- 3- [COMBINE strings into a single string "STR"]

$$STR = S1 \parallel " " \parallel S2 \parallel " " \parallel S3$$
- 4- END

Program Code 4.3

```
// Program that combines three strings into a single empty string
#include <iostream.h>
#include <conio.h>

class strcon
{
    public:

        // member function to concatenate strings
        combine(char s1[], char s2[], char s3[], char res[])
        {
            int i = 0, r = 0;
            for(i = 0; s1[i]!='\0'; i++, r++)
                res[r] = s1[i];
            for(i = 0; s2[i]!='\0'; i++, r++)
                res[r] = s2[i];
            for(i = 0; s3[i]!='\0'; i++, r++)
                res[r] = s3[i];
            res[r] = '\0';
        }
};

void main(void)
{
    strcon obj;
    char str1[20], str2[20], str3[20], str[80];
    clrscr();
    cout<<"Enter first string : ";
    cin>>str1;
    cout<<"Enter second string : ";
    cin>>str2;
    cout<<"Enter third string : ";
    cin>>str3;
```

```

obj.combine(str1, str2, str3, str);
cout<<"Combined string = "<<str;
getch();
} //end of main()

```

Output of the program

Enter first string : XYZ

Enter second string : ABC

Enter third string : MNO

Combined string =XYZABCMNO

4.2.4 Extracting Substring from a String

The process of taking out a substring from a given string is called *string extraction*. This operation is required when a part of a string is to be processed. Three types of information are required to extract a substring from a given string:

- ★ Name of the string
- ★ Starting position of the substring
- ★ Length of the substring

This information is used in a function for extracting the substring. The general syntax of the function is as follows:

```
substr(string, initial, length)
```

In the above syntax:

- string** It indicates the given string from which the substring is to be extracted.
- initial** It indicates the starting position of the substring within the string.
- length** It indicates the length of the substring.

For example, to extract 2 letters from string "Pakistan" starting from the 4th position, the function is specified as:

```
substr("Pakistan", 4, 2)
```

The output of this function will be substring "is". We can also use a string variable in place of the string "Pakistan".

Algorithm - Extracting Substring

Write an algorithm that extracts parts of the string "WELCOME" and displays the output as given under:

```

W
WE
WEL
.....
WELCOME

```


- 1- START
- 2- STR = "WELCOME"
- 3- C = 1
- 4- REPEAT step-5 WHILE C <= 7
- 5- PRINT substr(STR, 1, C)
[End of Step-4 Loop]
- 6- END

Program Code 4.4

```
// Program that extracts parts of a string
#include <iostream.h>
#include <conio.h>

class strextract
{
    private:
        char str[20];
    public:

        // member function to extract string
        substr(char st1[], int loc, int s)
        {
            for(int i = loc-1; i<=s; i++)
            {
                str[i] = st1[i];
                str[i+1] = '\0';
            }
            cout<<str<<endl;
        }
};

void main(void)
{
    strextract obj;
    clrscr();
    for(int c = 0; c<7; c++)
        obj.substr("WELCOME", 1, c);
    getch();
} //end of main()
```

Program's Output

```
W
WE
WEL
WELC
WELCO
WELCOM
WELCOME
```

4.2.5 Pattern Matching

In web applications, for example, while taking an email address from a person, we need to ensure that the given address contains @ sign at some position in the email address. For operations like these, we use Pattern Matching. The process of finding a substring within a string

is called *pattern matching*. The length of the substring to be matched must be less than or equal to the length of the string in which pattern matching will be performed. The most commonly used pattern matching algorithms are as follows:

- i) Naive String Matcher
- ii) Rabin-Karp Algorithm
- iii) Knuth-Morris-Pratt Algorithm

4.2.5.1 Naive String Matcher

Naive string matcher is also known as the *Brute Force string matching algorithm*. It is the simplest method among the other pattern-matching algorithms. It matches the pattern (i.e. given substring) to each substring of the same length in the given string. This approach is easy to understand and implement but it can be too slow in some cases. It is helpful for smaller texts. The general function used for naive pattern matcher is as follows:

`naive(string, substring)`

In this syntax:

string It indicates the main string within which the given substring is to be searched.

Substring It indicates the substring to be matched/searched.

The parameters may be either string constants or string variables or a mix of these. If the substring or pattern is not found in the string, then a zero value is returned.

The starting position of the pattern in the main string is called the *cursor position*. The cursor position indicates the left-most character of the substring in the main string. Suppose STR = "XYZABCXYZ" and a substring YZ is to be searched, then the function is specified as:

`naive(STR, "YZ");`

It will return cursor position 2 i.e. "YZ" starts at position 2 within the main string. Similarly, `naive(STR, "YZB")` will return 0 as cursor position because the substring "YZB" does not exist in the string STR.

The above pattern (i.e. "CX") is compared with each substring of STR, starting from left to right until the given pattern is matched. The comparison is made as explained below:

1. Compare "XY" to "CX"
2. Compare "YZ" to "CX"
3. Compare "ZA" to "CX"
4. Compare "AB" to "CX"
5. Compare "BC" to "CX"
6. Compare "CX" to "CX"
7. Compare "XY" to "CX"
8. Compare "YZ" to "CX"

X	Y	Z	A	B	C	X	Y	Z
C	X							
	C	X						
		C	X					
			C	X				
				C	X			
					C	X		
						C	X	

In this example, the pattern is matched at step-6. The cursor position is returned and the searching process is terminated.

For pattern matching, each substring of STR is compared with the pattern "CX" character by character. The total number of substrings in a string to compare a pattern is computed as:

$$\text{Number of Substrings} = \text{String_Length} - \text{Pattern_Length} + 1$$

In the above example, the length of the string STR is 9, and the length of the pattern is 2. Thus, the total number of substrings to be compared with "CX" is computed as:

$$\text{Number of Substrings} = 9 - 2 + 1 = 8$$

Algorithm – Naïve String Matcher

Write an algorithm that searches a substring within a given string using Naive string matcher.

```
1-   START
2-   INPUT string into variable STR
3-   INPUT Substring into variable SUBSTR
4-   FIND length of STR and store result in LEN
5-   FIND length of SUBSTR and store result in SUBLEN
6-   C = 0
7-   REPEAT Step-8 TO 10 WHILE C <= LEN - SUBLEN
8-   REPEAT Step-9 to 10 FOR I = 0 TO SUBLEN
9-   IF I = SUBLEN THEN
        PRINT "Pattern matched at position =", C+1
        EXIT
    END IF
10-  IF SUBSTR[I] = STR[C+I] THEN
        C = C + 1
        GO TO STEP-7
    END IF
    [End of step-8 inner loop]
    [End of step-7 upper loop]
11-  PRINT "Pattern not matched"
12-  END
```

Program Code 4.5

// Program that searches a substring within a string using Naive string matcher

```
#include <iostream.h>
#include <conio.h>
#include <string.h>

class pattern
{
    private:
        char str[30], substr[30];
        int len, sublen;
    public:
        void input(void);
        naive(void);
};
```

```
void main(void)
```

```
{
    pattern obj;
    clrscr();
    obj.input();
    obj.naive();
    getch();
}
```

```
//end of main()
```

```
// member function to input string and substring
```

```
void pattern::input(void)
```

```
{
    cout<<"Enter a string: ";
    cin>>str;
    cout<<"Enter a substring: ";
    cin>>substr;
}
```

```
//end of input()
```

```
// member function to match pattern
```

```
pattern::naive(void)
```

```
{
    len = strlen(str);
    sublen = strlen(substr);

    int i, c;
    for( c = 0; c<=len - sublen; c++)
    {
        for(i = 0 ;i<=sublen; i++)
        {
            if(i==sublen)
            {
                cout<<"Pattern matched at position = "<<c+1;
                return 0;
            }
        }
    }
}
```

Program's Output

Enter a string: Pakpattan

Enter a substring: pat

Pattern matched at position = 4


```
        }  
        if(substr[i] != str[c+i])  
            break;  
    }  
    }  
    cout<<"Pattern not matched ";  
    return 0;  
} //end of naive()
```

Complexity Analysis of Naive String Matcher

Naive string matcher algorithm uses a nested loop. The upper loop slides the window on the input string to right on each iteration, while the inner loop compares the current shift of the input string with the pattern that is to be found. The average-case time complexity for naive string matcher is $O(n+m)$, where n is the size of the input string, and m is the length of the pattern string. For the worst-case, the time complexity is $O(n*m)$. It does not occupy extra memory space to perform the operation. Following table shows the complexities of naive string matcher:

Time Complexity		Space Complexity
Average-Case	Worst-Case	Worst-Case
$\Theta(n+m)$	$O(n*m)$	None

4.2.5.2 Rabin-Karp Algorithm

Naive string matching algorithm slides the pattern one by one. After each slide, it checks characters at the current window one by one, and if all characters match then prints the match.

Rabin-Karp is another pattern matching algorithm. Like the Naive algorithm, the Rabin-Karp algorithm also checks the pattern by moving the window one by one, but without checking all characters for all cases. Rabin Karp algorithm matches the hash value of the pattern with the hash value of the current substring of the input string. When the hash values are matched, then only it starts matching individual characters. This procedure makes the algorithm more efficient. Rabin Karp algorithm needs to calculate hash values for the following strings:

- Pattern itself
- All the substrings of the given string, each of length "m"

Since we need to efficiently calculate hash values for all the substrings of size "m" of the given string using the hash function. The hash function suggested by Rabin and Karp calculates an integer value. For example, if all possible characters are from 1 to 10, the numeric value of "122" will be 122. The number of possible characters is higher than 10 (256 in general) and pattern length can be large. So the numeric values cannot be practically stored as an integer. Therefore, the numeric value is calculated using modular arithmetic to make sure that the hash values can be stored in an integer variable.

Algorithm – Rabin-Karp Algorithm

Write an algorithm that searches a substring within a given string using the Rabin-Karp algorithm.

```

1-   START
2-   INPUT string into variable STR
3-   INPUT substring into variable SUBSTR
4-   Define D = 256 (Maximum Value)
5-   FIND length of STR and store result in LEN
6-   FIND length of SUBSTR and store result in SUBLEN
7-   Q = 101, H=1, T=0, P=0
8-   REPEAT Step-9 FOR J = 0 To SUBLEN - 1
9-   H = (H*D)%Q
10-  REPEAT Step-11 to 12 FOR J = 0 To SUBLEN
11-  P = (D * P + SUBSTR[J]) % Q
12-  T = (D * T + STR[J]) % Q
13-  REPEAT Step-14 TO 19 WHILE C = 0 To LEN - SUBLEN
14-  IF P = T THEN
15-    REPEAT Step-16 to 18 FOR I= 0 TO SUBLEN
16-    IF I = SUBLEN THEN
17-      PRINT "Pattern matched at position =", C+1
18-      EXIT
19-    END IF
20-  IF SUBSTR[I]!=STR[C+I] THEN
21-    C = C + 1
22-    GO TO STEP-13
23-  END IF
24- END IF
25- [End of step-15 inner loop]
26- IF C < LEN - SUBLEN THEN
27-   T = (D * (T - STR[C] * H) + STR[C + SUBLEN]) % Q;
28-   IF T < 0 THEN
29-     T = T + Q
30-   END IF
31- END IF
32- [End of step-13 upper loop]
33- PRINT "Pattern not matched"
34- END

```

Program Code 4.6

// Program that searches a substring within a string using Rabin_Karp algorithm


```
#include <iostream.h>
#include <conio.h>
#include <string.h>
#define d 256

class pattern
{
    private:
        char str[30], substr[30];
        int len, sublen, q;
    public:
        void input(void);
        rabin_karp(void);
};

void main(void)
{
    pattern obj;
    clrscr();
    obj.input();
    obj.rabin_karp();
    getch();
} //end of main()

// member function to input string and substring
void pattern::input(void)
{
    cout<<"Enter a string: ";
    cin>>str;
    cout<<"Enter a substring: ";
    cin>>substr;
} //end of input()

// member function to match pattern
pattern::rabin_karp(void)
{
    len = strlen(str);
    sublen = strlen(substr);

    q = 101; // Any Prime number
    int p = 0; // hash value for substring
    int t = 0; // hash value for string
    int h = 1;

    // The value of h would be "pow(d, sublen-1)%q"
    for (int j = 0; j < sublen-1; j++)
        h = (h*d)%q;
```

Program's Output

Enter a string: Pakpattan

Enter a substring: Pak

Pattern-matched at position = 1

```

        // Calculate the hash value of substring and first window of string
        for (j = 0; j < sublen; j++)
        {
            p = (d*p + substr[j])%q;
            t = (d*t + str[j])%q;
        }
        int i, c;
        for (c = 0; c <= len - sublen; c++)
        {
            if (p == t)
            {
                for (i = 0; i <= sublen; i++)
                {
                    if (i == sublen)
                    {
                        cout << "Pattern matched at position = " << c + 1;
                        return 0;
                    }
                    if (substr[i] != str[c + i])
                        break;
                }
            }

            // Calculate hash value for next window of text: Remove leading digit, add trailing digit
            if (c < len - sublen)
            {
                t = (d*(t - str[c]*h) + str[c + sublen])%q;

                // We might get negative value of 't', converting it to positive
                if (t < 0)
                    t = (t + q);
            }
        }
        cout << "Pattern not matched ";
        return 0;
    } //end of rabin_karp()

```

Complexity Analysis of Rabin-Karp Algorithm

Rabin-Karp string matching algorithm uses hashing. Once the hash value matches for the pattern and a window of the input string, then it verifies each character of the window with the pattern to make sure that the match is true or valid. For the best and average case of the Rabin-Karp algorithm, all the matches are valid. It means whenever the hash value matches, the characters of the pattern and window also matches. In this case, the time complexity of the Rabin-Karp algorithm is $\Theta(n+m)$ where "n" is the length of the input string and "m" is the length of the matching pattern.

In the worst case of the Rabin-Karp algorithm, it gives bogus hits. It means that the hash value matches, but the window does not match with the pattern. In this case, "m" characters will be checked for "n" times and time complexity is $O(n*m)$. The worst-case of the Rabin-Karp algorithm can occur when the prime number chosen is very small.

Rabin-Karp algorithm takes constant space in memory. Following table shows the time and space complexity of the Rabin-Karp algorithm.

Time Complexity		Space Complexity
Average-Case	Worst-Case	Worst-Case
$\Theta(n+m)$	$O(nm)$	$O(1)$

4.2.5.3 Knuth-Morris-Pratt Algorithm (KMP)

KMP was the first linear-time algorithm for string matching. This algorithm uses the degenerating property of the pattern. It means that each time when a match between the pattern and a window of the input string fails, at that time some of the characters of the input string of the next window are already known. The algorithm uses that information to avoid matching the characters that are already matched.

Suppose the input string is "AAAAABAAABA" and pattern is "AAAA". This algorithm compares the input string with the given pattern as follows:

Shift 1: Compare the first window of the input string with a pattern.

First window of input string = "AAAA"

Pattern = "AAAA"

Match found. This is the same as a naive string matcher.

Shift 2: Compare the next window of the input string with a pattern.

Second window of input string = "AAAA"

Pattern = "AAAA"

This is where KMP does optimization over Naive. In this second window, only compare the fourth A of the pattern with the fourth character of current window of the input string to decide whether current window matches or not. Since the first three characters are already known and matched, hence, skip matching the first three characters.

An important question arises how many characters to be skipped. To know this, preprocess pattern and prepare an integer array LPS which tells the count of characters to be skipped.

Algorithm – Knuth-Morris-Pratt Algorithm

Write an algorithm that searches a substring within a given string using the KMP algorithm.

1- START

```

2-   INPUT string into variable STR
3-   INPUT Substring into variable SUBSTR
4-   FIND length of STR and store result in LEN
5-   FIND length of SUBSTR and store result in SUBLEN
6-   LPS = CALL COMPUTE_PREFIX_FUNCTION()
7-   J = 0
8-   REPEAT Step-9 to 10 WHILE I = 0 To LEN
9-       IF SUBSTR[J] = STR[I] THEN
           J++
           I++
       END IF
10-  IF J = SUBLEN THEN
       PRINT "Pattern matched at position =", I-(J-1)
       J = LPS [J-1]
  ELSE IF I < LEN AND SUBSTR[J] != STR[I] THEN
       IF J != 0 THEN
           J = LPS [J-1]
       ELSE
           I++
       END IF
  END IF
  [End of step-8 while loop]
  EXIT

11-  COMPUTE_PREFIX_FUNCTION()
       PRE_LEN = 0
       LPS[0] = 0
12-  REPEAT Step-13 WHILE I = 1 To SUBLEN
13-  IF SUBSTR[I] = SUBSTR[PRE_LEN] THEN
       LPS[I] = PRE_LEN ++
       I ++
  ELSE IF PRE_LEN != 0
       PRE_LEN = LPS[PRE_LEN - 1]
  ELSE
       LPS[I] = 0
       I ++
  END IF
  END IF
  EXIT
14-  END

```


Program Code 4.7

```
// Program that searches a substring within a string using KMP algorithm
```

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
#include <string.h>
```

```
class pattern
```

```
{
```

```
private:
```

```
char str[30], substr[30], lps[30];
```

```
int len, sublen, q;
```

```
public:
```

```
void input(void);
```

```
compute_prefix_function(void);
```

```
kmp(void);
```

```
};
```

```
void main(void)
```

```
{
```

```
pattern obj;
```

```
clrscr();
```

```
obj.input();
```

```
obj.kmp();
```

```
getch();
```

```
} //End of main
```

```
void pattern::input(void)
```

```
{
```

```
cout<<"Enter a string: ";
```

```
cin>>str;
```

```
cout<<"Enter a pattern: ";
```

```
cin>>substr;
```

```
} //end of input()
```

```
pattern::kmp(void)
```

```
{
```

```
len = strlen(str);
```

```
sublen = strlen(substr);
```

```
compute_prefix_function(); // Preprocess the pattern (calculate lps[] array)
```

```
int i = 0, j = 0;
```

```
while (i < len)
```

```
{
```

```
if (substr[j] == str[i])
```

```
{
```

```
j++;
```

```
i++;
```

```
}
```

Program's Output

```
Enter a string: Pakistan
```

```
Enter a substring: kis
```

```
Pattern matched at position = 3
```

```

        if (j == sublen)
        {
            cout<<"pattern matched at position = "<< i-(j-1);
            j = lps[j - 1];
        }
        // mismatch after j matches
        else if (i < len && substr[j] != str[i])
        {
            if (j != 0)
                j = lps[j - 1];
            else
                i = i + 1;
        }
    }
    return 0;
} //End of KMP

pattern::compute_prefix_function(void) // Fills lps[] for given pattern pat[0..M-1]
{
    int pre_len = 0; // length of the previous longest prefix suffix
    lps[0] = 0; // lps[0] is always 0
    int i = 1;
    while (i < sublen) // the loop calculates lps[i] for i = 1 to M-1
    {
        if (substr[i] == substr[pre_len])
        {
            pre_len++;
            lps[i] = pre_len;
            i++;
        }
        else
        {
            if (pre_len != 0)
            {
                // Also, note that we do not increment i here
                pre_len = lps[pre_len - 1];
            }
            else
            {
                lps[i] = 0;
                i++;
            }
        }
    }
    return 0;
} // End of compute_prefix_function

```


Complexity Analysis of Knuth-Morris-Pratt algorithm

Knuth-Morris-Pratt string matching algorithm matches the characters from left to right. When a pattern has a sub-pattern appears more than once in the sub-pattern, it uses that property to improve the time complexity, also for in the worst-case. The algorithm uses a while loop in "kmp()" function and a while loop in "compute_prefix_function()" function. Thus, the KMP algorithm is a linear time algorithm.

KMP algorithm takes space to store "lps" array of size equal to pattern length in memory. Therefore, its space complexity is $O(m)$, where m is the length of the pattern. Following table shows the time and space complexity of the KMP algorithm.

Time Complexity		Space Complexity
Average-Case	Worst-Case	Worst-Case
$\Theta(n)$	$O(n)$	$O(m)$

4.2.6 Insertion Operation

In insertion operation, a substring is added (or inserted) at a specified location in a string. Suppose there is a function "insert()" used to insert a substring in a given string. The general syntax of this function is:

`insert(string, position, substring)`

In this syntax:

- string** It indicates the given string into which substring or text is to be inserted.
- position** It indicates the position where the substring is to be inserted.
- substring** It indicates the substring that is to be inserted into the string.

For example, to insert "pat" in a string "Paktan" at position 4, the *insert* function will be written as:

`insert("Paktan", 4, "pat");`

The above function will return the string "Pakpattan" as an output.

During insertion, the *extraction* and *concatenation* operations are used. In extraction, the string into which the substring is to be inserted is divided into the following two parts:

- ★ The first part consists of the text of the given string, starting from the first character up to the specified position where the substring is to be inserted.
- ★ The second part consists of the remaining text of the string i.e. from the specified position up to the last character of the string.

The text that is to be inserted is concatenated with the first part of the string. The second part of the original string is concatenated with the previously obtained string.

Algorithm – Inserting Substring Into String

Write an algorithm that inserts a substring into a given string at a specific location.

- 1- START
- 2- INPUT string into variable STR
- 3- INPUT substring into variable SUBSTR
- 4- INPUT location to insert SUBSTR into STR in variable LOC
[Extract first part of string from location 1 to LOC into ST1]
- 5- ST1 = SUBSTRING[STR, 1, LOC]
[Extract remaining part of the string from LOC to end of the string into ST2]
- 6- ST2 = SUBSTRING [STR, LOC, LENGTH(STR)]
[Combine the strings, ST1, SUBSTR and ST2]
- 7- RESULT = ST1 + SUBSTR + ST2
- 8- PRINT RESULT
- 9- END

Program Code 4.8

// Program that inserts a substring into a given string

#include <iostream.h>

#include <conio.h>

class insert_string

{

private:

char str[75], st1[25], st2[25], substr[25];

int loc, len;

public:

void input(void);

void strlen(void);

void extract(void);

void insert(void);

};

void main(void)

{

insert_string obj;

clrscr();

obj.input();

obj strlen();

obj.extract();

obj.insert();

getch();

} //end of main()

// member function to input strings and location

void insert_string::input(void)

Program's Output

Enter string : America

Enter substring to insert : NO

Enter location in string to insert : 5

Resultant string = AmerNOica


```
{
    cout<<"Enter string : ";
    cin>>str;
    cout<<"Enter substring to insert : ";
    cin>>substr;
    cout<<"Enter location in string to insert : ";
    cin>>loc;
    loc--; // because first position starts from 0
} //end of input()

// member function to find string length
void insert_string::strlen(void)
{
    for(int i = 0; str[i]!='\0'; i++)
        len = i;
} //end of strlen()

// member function to divide string into two substrings
void insert_string::extract(void)
{
    int i;

    // copy string str from position 1 to loc into st1
    for(i = 0; i<loc; i++)
        st1[i] = str[i];
    st1[i] = '\0';

    // copy remaining part of string str from loc to end of string into st2
    for(i = loc; i<=len; i++)
        st2[i-loc] = str[i];
    st2[i-loc] = '\0';
} //end of extract()

// member function to combine three strings
void insert_string::insert(void)
{
    int c, i;
    c = 0;

    // copy st1 to str
    for(i = 0; st1[i]!='\0'; i++, c++)
        str[c] = st1[i];

    // copy substr to str
    for(i = 0; substr[i]!='\0'; i++, c++)
        str[c] = substr[i];

    // copy st2 to str
    for(i = 0; st2[i]!='\0'; i++, c++)
```

```

        str[c] = st2[i];
    str[c] = '\0';
    cout<<"Resultant string = "<<str;
} //end of insert()

```

4.2.7 Deletion Operation

During text processing, sometimes we need to find and delete a specific word or string in our text. This happens in cases where an extra word or string is mistakenly written, one or more times. For this purpose, we use the *deletion* operation.

In deletion operation, a substring is removed from a specified position in a given string. Suppose we use a function "*delete*" to remove a substring from a given string. This function has the following syntax;

```
delete(string, position, size)
```

In this syntax:

- string** It indicates the string from which the part of the text is to be deleted.
- position** It indicates the starting position of the substring within the string from which the substring is to be removed.
- size** It indicates the size of the substring to be removed.

Suppose there is a string STR = "CDEXYZ" and we want to delete the substring "EX" from the string STR. The starting position of "EX" in STR is 3 and its length is 2. Therefore the *delete* function is written as:

```
delete(STR, 3, 2);
```

We can see that the deletion operation is identical to the insertion operation. The only difference is that in deletion, the extracted substring is omitted when the two parts of the original string are again concatenated.

Algorithm – Deleting Substring from String

Write an algorithm that deletes a substring from a given string at a specific location.

- 1- START
- 2- INPUT string into STR
- 3- INPUT substring into SUBSTR
- 4- INPUT location of the substring to delete in LOC
- 5- INPUT length of the substring to delete in SUBLEN
- 6- Compute the length of string and store result in LEN
 [Extract first part of the string from 1 to LOC to ST1]
- 7- ST1 = SUBSTRING[STR, 1, LOC]
 [Extract second part from LOC up to end of the string to ST2]

- 8- ST2 = SUBSTRING [STR, LOC, LENGTH(STR)]
[Extract part of string ST2 from SUBLEN up to end and store into ST3]
- 9- ST3 = SUBSTRING [ST2, SUBLEN, LENGTH(ST2)]
[Combine the strings ST1 and ST3]
- 10- RESULT = ST1 + ST3
- 11- PRINT RESULT
- 12- END

Program Code 4.9

/ Program that deletes part of a given string

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
class delete_string
```

```
{
```

```
private:
```

```
char str[100], st[5], st2[50];
```

```
int loc, len, sublen;
```

```
public:
```

```
void input(void);
```

```
void length(void);
```

```
void extract(void);
```

```
void del_substring(void);
```

```
void main(void)
```

```
{
```

```
delete_string obj;
```

```
clrscr();
```

```
obj.input();
```

```
obj.length();
```

```
obj.extract();
```

```
obj.del_substring();
```

```
getch();
```

```
} //end of main()
```

```
// member function to input data
```

```
void delete_string::input(void)
```

```
{
```

```
cout<<"Enter string : ";
```

```
cin>>str;
```

```
cout<<"Enter length of substring to delete : ";
```

```
cin>>sublen;
```

```
cout<<"Enter location to delete substring : ";
```

Program's Output

Enter string : AMERICA

Enter length of substring to delete : 3

Enter location to delete substring : 2

Resultant string = AICA

```

        cin>>loc;
        loc--;          // because first position starts from 0
    } //end of input()

    // member function to find string length
    void delete_string::length(void)
    {
        for(int i = 0; str[i]!='\0'; i++)
            len = i;
    } //end of length()

    // member function to divide string str into two substrings st1 & st2
    void delete_string::extract(void)
    {
        int c1, i;
        // copy string str from 0 to loc into st1
        for(i = 0; i<loc; i++)
            st1[i] = str[i];
        st1[i] = '\0';

        // copy remaining part of string str into st2
        for(i = loc; i<=len; i++)
            st2[i-loc] = str[i];
        st2[i-loc] = '\0';
    } //end of extract()

    // member function to delete substring
    void delete_string::del_substring(void)
    {
        int c = 0, i;

        // copy st1 to str
        for(i = 0; st1[i]!='\0'; i++, c++)
            str[c] = st1[i];
        str[c] = '\0';

        // copy st2 from sublen to end of string to str
        for(i = sublen; st2[i]!='\0'; i++, c++)
            str[c] = st2[i];
        str[c] = '\0';
        cout<<"Resultant string = "<<str;
    } //end of del_substring()

```

4.2.8 Replacement Operation

Finding and replacing a specific word or string in documents is yet another important operation. In word processing applications like Microsoft Word, Find & Replace is considered to be a necessary feature.

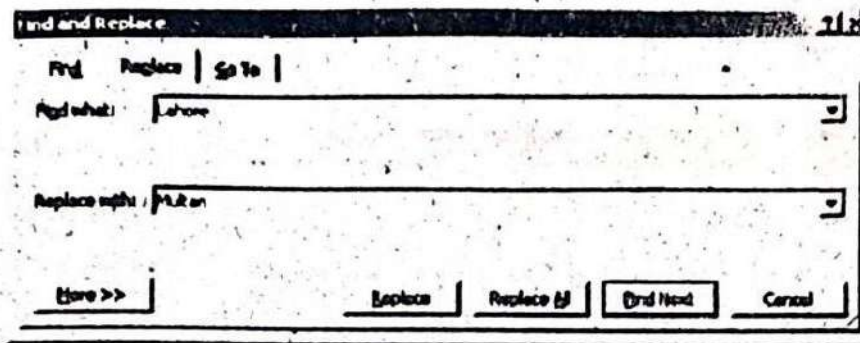


Figure: Find and Replace dialog box in MS Word

In replacement operation, a substring is searched within a given string and, if found, is replaced with a new substring. Suppose we use a function “replace” to replace a substring in a given string. The general syntax of the *replace* function is as follows:

`replace(string, substr1, substr2)`

In this syntax:

string It indicates the string in which the substring is to be searched and replaced.

substr1 It indicates the substring that is to be searched in the string.

substr2 It indicates a new substring or text that is to be replaced with the substr1 in the string.

Suppose we have a string `STR = "PAKPATTAN"`. To replace “PAT” with “IS” in `STR`, the *replace* function will be written as:

`replace (STR, "PAT", "IS");`

The contents of `STR` after replacement will be “PAKISTAN”. It can be observed that the replacement operation consists of the following three operations:

- ★ Searching or indexing operation to search `substr1`.
- ★ Deletion operation to delete the `substr1` from the given string.
- ★ Insertion operation to insert the new value in place of `substr1`.

If there is a string `STR = "I am Pakistan"`, then to replace “am” with text “Love” in `STR`, the following steps will be performed:

1. `LOC = search(STR, "am");`
Search “am” within the string `STR`. Return its position and store the position into `LOC`. In this example, the value 3 will be assigned to `LOC`.
2. `STR = delete(STR, LOC, length("am"));`
Delete “am” from `STR`. The resulting string is stored in the string variable `STR`, i.e. `STR = "I Pakistan"`.
3. `Insert(STR, LOC, "Love");`

- Insert substring "Love" at location LOC, i.e. at location 3 in string STR. The contents of the string after insertion are STR = "I Love Pakistan".

Algorithm – Searching & Replacing Substring In a String

Write an algorithm that searches a substring in a given string and replaces it with new one.

- 1- START
- 2- INPUT string in STR
- 3- INPUT substrings in SUBSTR1 and SUBSTR2
[Search SUBSTR1 in STR]
- 4- LOC = search(STR, SUBSTR1)
- 5- IF LOC > 0 THEN
 STR = DELETE(STR, LOC, length(SUBSTR1))
 INSERT(STR, LOC, SUBSTR2)
ELSE
 PRINT "Substring not found"
[End of IF Structure]
- 6- END

Program Code 4.10

// Program that searches a substring within a string and replaces it with new one

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
class replace
```

```
{
```

```
private:
```

```
char str[80], substr1[40], substr2[40], st1[40], st2[40];  
int len, sublen1, total;
```

```
public:
```

```
int search(void);  
void del(int);  
void insert(void);
```

```
};
```

```
void main(void)
```

```
{
```

```
replace obj;
```

```
int loc;
```

```
clrscr();
```

```
loc = obj.search();
```

```
if(loc > 0)
```

```
{
```

```
obj.del(loc);
```

```
obj.insert();
```



```
    }
    else
        cout<<"Substring not found";
    getch();
} //end of main()

// member function to match pattern
int replace::search()
{
    int i, c;
    clrscr();
    cout<<"Enter main string : ";
    cin>>str;
    cout<<"Enter a substring to search : ";
    cin>>substr1;

    // find length of main string
    for(i = 0; str[i]!='\0'; i++)
        len = i + 1;
    // find length of substring
    for(i = 0; substr1[i]!='\0'; i++)
        sublen1 = i + 1;
    total = len - sublen1 + 1;
    for( c = 0; c<=total; c++)
    {
        for(i = 0 ;i<=sublen1; i++)
        {
            if(i==sublen1)
                return c;
            if(substr1[i]!= str[c+ i])
                break;
        }
    }
    c = -1;
    return c;
} //end of search()

// member function to delete a searched substring from main string
void replace::del(int pos)
{
    int c1, i;

    // copy first part of main string from 0 to pos into st1
    for(i = 0; i<pos; i++)
        st1[i] = str[i];
    st1[i] = '\0';
```

```

        // copy second part of main string from searched substring up to end into st2
        for(i = 0, c1 = pos + sublen1; str[c1]!='\0'; i++, c1++)
            st2[i] = str[c1];
        st2[i] = '\0';
    } //end of del()

    // member function to combine substrings st1, substr2, and st2
    void replace::insert(void)
    {
        int i, c;
        cout<<"Enter a substring to replace : ";
        cin>>substr2;

        // copy st1 into str
        for(i = 0, c = 0; st1[c]!='\0'; i++, c++)
            str[c] = st1[i];

        // combine substr2 with str
        for(i = 0; substr2[i]!='\0'; i++, c++)
            str[c] = substr2[i];

        // combine sr2 with str
        for(i = 0; st2[i]!='\0'; i++, c++)
            str[c] = st2[i];
        str[c] = '\0';
        cout<<"Main string after replacing text: "<<str;
    } //end of insert()

```

Output of the program:

Enter main string : KASHMIR
 Enter a substring to search : SH
 Enter a substring to replace : PAKISTAN
 Main string after replacing text: KAPAKISTANMIR

Source Code of Programs

Source code of programs given in "DATA STRUCTURES USING C++" can be received by sending an e-mail to "pakman_series@hotmail.com".

Short Answers to the Questions

What is a string?

A string is a sequence of characters. It may include alphabetic letters (a-z), numeric digits (0-9) and various special characters such as +, -, *, ^, /, #, \$ etc.

What is the difference between the fixed-length string and variable-length string?

In fixed-length string, the amount of memory required for storing the string is fixed at the time the string variable is declared. Its length remains fixed throughout the execution of the program.

In variable-length string, the amount of memory required for storing a string is not fixed at the time of declaration of the string variable. However, the size can be changed during the execution of the program. Most modern programming languages use variable-length strings.

What is string concatenation?

The process of combining the contents of two or more strings into a single string is called *string concatenation*.

What is meant by pattern matching?

The process of finding a substring within a string is called *pattern matching*. The length of the substring to be matched must be less than or equal to the length of the string in which pattern matching will be performed.

EXERCISE

Q.1 Select the correct answer from the multiple choices.

1. A sequence of characters or symbols is called:
(a) data (b) information
(c) data structure (d) string
2. A string is terminated by:
(a) dollar symbol (b) Null character
(c) slash character (d) None of these
3. In C++, null character is indicated by:
(a) 0 (b) \0
(c) /0 (d) \\0
4. A string with zero length is called:
(a) empty string (b) null string
(c) nil string (d) Both (a) & (b)

5. The string which has a fixed length and uses the same amount of memory is called:
 (a) variable-length string (b) constant string
 (c) static string (d) fixed-length string
6. Which type of string is more dynamic and powerful type of string?
 (a) variable-length string (b) constant string
 (c) static string (d) fixed-length string
7. Which type of string is declared without specifying the string size?
 (a) fixed-length string (b) constant string
 (c) static string (d) variable-length string
8. Combining the contents of two or more strings into a single string is called:
 (a) string combination (b) string concatenation
 (c) string merging (d) string extraction
9. Taking out a substring from a given string is called:
 (a) string combination (b) string concatenation
 (c) string merging (d) string extraction
10. Finding a substring within a string is called:
 (a) string finding (b) string concatenation
 (c) string searching (d) pattern matching

Answers of MCQs

01 (d)	02 (b)	03 (b)	04 (d)	05 (d)	06 (a)	07 (d)	08 (b)	09 (d)	10 (d)
--------	--------	--------	--------	--------	--------	--------	--------	--------	--------

- Q.2 What is a string? Explain different methods of representing strings in the computer's memory.
- Q.3 Explain different types of strings with examples.
- Q.4 Write an algorithm to count the total number of alphabets and numeric digits (if any) in a string.
- Q.5 Write a program to count the total number of capital letters in a string.
- Q.6 What is a sub-string? Discuss different operations where sub-strings are used.
- Q.7 Write a program to copy one string into another and display the resulting string as an output.
- Q.8 Write a program to convert a string from lowercase to uppercase characters.

STACKS

5.1 STACK

A stack is a linear data structure in which items are accessible only from one end. This end is called *Top* of the stack. When a new item is inserted (added) into the stack, it is placed on top of all the previously added items. The items are removed in the reverse order of which they are added i.e. top of the item is removed first. Therefore, the items in the stack are stored and retrieved in "Last In First Out - (LIFO)" manner. It means that the item inserted in the last is retrieved first. For this reason, the stack is also known as *Last-In, First-Out (LIFO)* data structure.

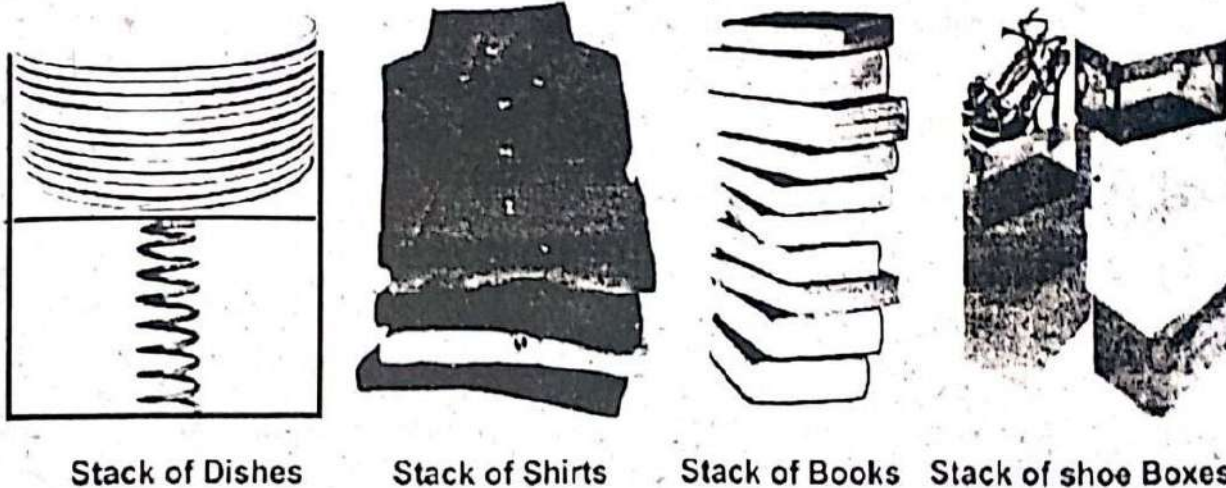


Figure 5.1: Examples of stacks in the real world

The above pictures show different stacks in the real world. We can observe that in any of these stacks, an item can only be removed from the top, or a new item can be placed on the top only.

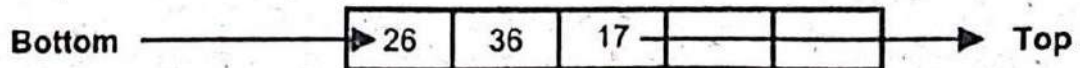
The most common example of a stack is a dish rack. A dish rack is a spring-loaded device that holds dishes in such a manner that only the top dish is visible. It is used in the following style:

- ★ A clean dish is placed on top of the stack. This forces the spring down and only the newly added dish is available for access.
- ★ When a clean dish is needed, only the top one is removed. This causes the spring to release so that the second dish becomes available now.
- ★ The last dish cleaned is the first one to be used.

5.1.1 Implementation of Stack

Stacks can be implemented by using an array or linked list. An array provides a simple way to implement any stack. However, it allows only fixed data items to be added to the stack. It is because the size of a static array cannot be changed during the execution of a program. The other way to implement the stack is by using a linked list. We know that in linked lists, memory is allocated dynamically to store data items into a stack. So any number of data items can be added into the stack providing more flexibility to the programmers.

The following figure shows a stack implementation by using an array. It has a capacity of 5 elements; however, three items are stored in it.



The top-most accessible position in the stack is called the *Top* of the stack and the first item's position is called *Bottom* of the stack.

⚠ Implementation of the stack is discussed in the chapter "Linked List - part-3".

5.1.2 Basic Features of Stack

Following are the basic features of the stack:

- 1- Stack is an ordered list of items with a similar data type.
- 2- Stack is a LIFO data structure (i.e. items are accessed in Last In First Out) manner.
- 3- The *PUSH* operation is used to insert a new element into the stack and *POP* operation is used to delete an element from the top of the stack. Please note that both insertion and deletion are allowed at only one end of the stack called *TOP*.
- 4- A stack is said to be in *OVERFLOW* state when it is completely full, whereas it is said to be in *UNDERFLOW* state if it is completely empty.

5.1.3 Stack Operations

As described above, two fundamental operations can be performed on the stack. These operations are called *PUSH* and *POP*. A brief description of these operations is given as under:

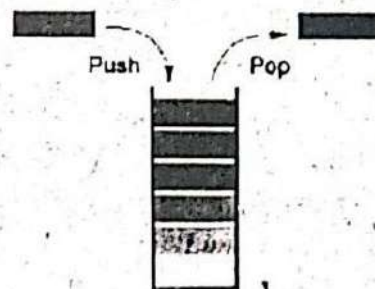


Figure 5.2: *PUSH and POP Operations*

5.1.3.1 PUSH Operation

When a new data item is added or inserted into the stack at its top position, the operation is called *PUSH operation*. Before pushing a data item into the stack, it is ensured that there is an available location in the stack to store the data item. If there is no room, no further value can be

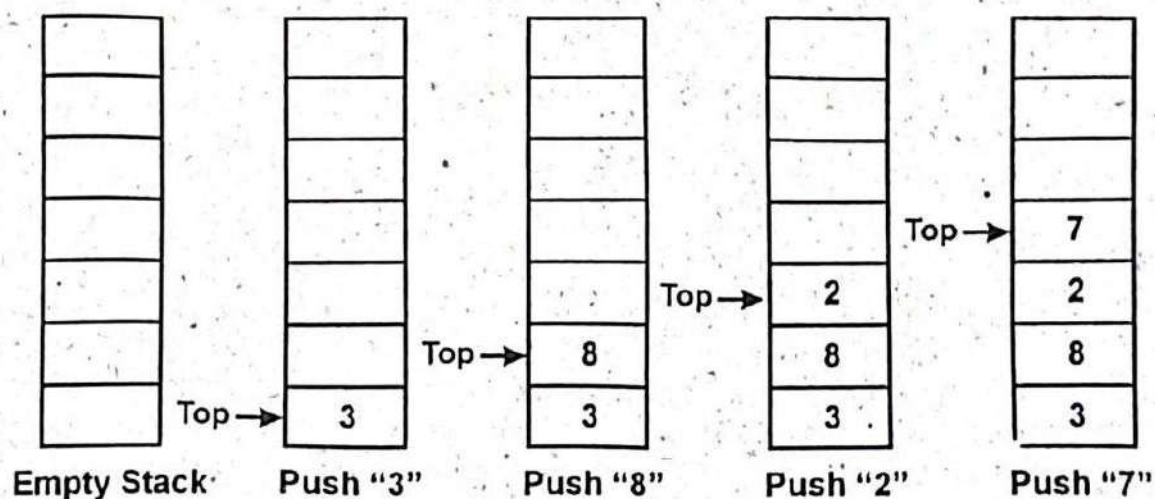
pushed onto the stack. This situation or condition is called "*stack overflow*". In this case, the "stack overflow" message is sent to the user.

The *PUSH* operation involves the following steps:

- 1- Check if the stack is full.
- 2- If the stack is full, produce an error message and exit the operation.
- 3- If the stack is not full, make an increment to the top to point the next empty space.
- 4- Add the new data element into the top position of the stack.
- 5- Exit the operation.

Example:

The sequence of *PUSH* operation in the stack is shown in the following diagrams:



In the beginning, the stack is empty. First of all, the number 3 is pushed into the stack. As a result, the number 3 comes at top of the stack. The pointer *Top* points at the top element. Each time a new item is pushed into the stack, the value of *Top* is incremented by 1. Then the number 8 is pushed into the stack and the value of *Top* is incremented by 1. Now see how numbers 3 and 8 are stacked. The number 8 is placed at the top of number 3 and the pointer *Top* moves one step upward. Similarly, numbers 2 and 7 are pushed one after the other.

5.1.3.2 POP Operation

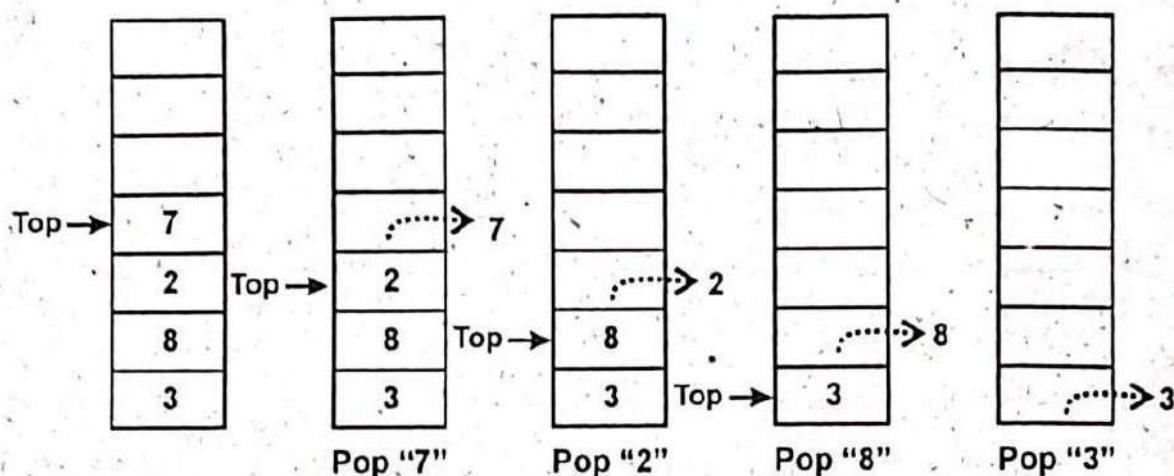
When a data item is removed from the stack, the operation is called *POP operation*. Before removing a data item from the stack, it is ensured that there is at least one item in the stack. If there is no data item, the stack is declared as "*empty*". This situation or condition is also called "*stack underflow*".

A *POP* operation involves the following steps:

- 1- Check if the stack is empty.
- 2- If the stack is empty, produce an error message and exit the operation.
- 3- If the stack is not empty, access the data element to which the Top is pointing.
- 4- Decrease the value of the Top by 1.
- 5- Exit the operation.

Example:

The working of *POP* operation in the stack is shown in the following diagrams:



The above sequence of figures shows the *POP* operation. First of all, the number 7 is popped. After this, the number 2 is popped. Now the number 8 is at the top. If we pop again, then number 8 is popped. Now the number 3 is at the top and this number is the last number in the stack. Pop again and the number 3 is popped. Now the stack is empty and there is no element in the stack (This condition of the stack is called *stack underflow*).

In both examples of *PUSH* and *POP* operations, we have seen that values are added and removed at the top of the stack. The first element pushed in the stack is the last to come out. Similarly, the last element pushed in the stack is the first to come out from the stack. Because the last element pushed in the stack is at the top which is removed when a *POP* operation is performed. That is why a stack is known as *LIFO* (Last In First Out) structure.

Procedure Algorithm – Pushing Item

Write a procedure sub-algorithm to push a data item *X* into a stack '*STK*' which can store *N* elements.

PUSH(STK, X)

- 1- [Check stack whether it has space or not]
IF TOP >= N THEN


```

        PRINT "Stack overflow"
        RETURN
    END IF
2-    [Insert value X into stack]
        TOP = TOP + 1
        STK[TOP] = X
3-    EXIT

```

Procedure Algorithm – Popping Item

Write a procedure sub-algorithm to remove a data item from a stack 'STK' which has N elements.

POP(STK, TOP)

```

1.    [Check stack if it is empty]
    IF TOP = 0 THEN
        PRINT "Stack is empty"
        RETURN
    END IF
2.    [Remove data item from stack]
    STK[TOP] = NULL
    TOP = TOP - 1
3.    RETURN

```

Program Code 5.1

```

// program to push and pop items in a stack
#include <iostream.h>
#include <conio.h>

```

```

class stack
{
    private:
        int top;
        int stk[5];
    public:
        stack(void) { top = -1; }

```

```

        // Prototypes or declarations of the functions of the class
        void push(int);
        void pop(void);
        void display(void);

```

```
};
```

```

void main(void)
{

```

```
stack st;
int n, opt, loop = 1;
while(loop)
{
    clrscr();
    cout<<"1- Pushing stack "<<endl;
    cout<<"2- Popping stack "<<endl;
    cout<<"3- Display stack "<<endl;
    cout<<"4- Exit "<<endl;
    cout<<"Enter your option [1-4] ? ";
    cin>>opt;
    switch(opt)
    {
        case 1:
            cout<<"Enter value to insert : ";
            cin>>n;
            st.push(n);
            break;
        case 2:
            st.pop();
            break;
        case 3:
            cout<<"Values in stack\n";
            st.display();
            break;
        case 4:
            loop = 0;
            break;
        default:
            cout<<"Invalid option";
    } //end of switch
} //end of while loop
} //end of main()

// member function to push value into stack
void stack::push(int x)
{
    if(top == 4)
    {
        cout<<"Stack overflow";
        getch();
        return;
    }
    top = top+1;
    stk[top] = x;
} //end of push()

// member function to pop value from stack
```



```
void stack::pop(void)
{
    int val;
    if(top == -1)
    {
        cout<<"Stack is empty";
        getch();
        return;
    }
    val = stk[top];
    stk[top] = NULL;
    top = top-1;
    cout<<"Value "<<val<<" is removed "<<endl;
    getch();
} //end of pop

// member function to display values of stack
void stack::display(void)
{
    if(top == -1)
    {
        cout<<"Stack is empty";
        getch();
        return;
    }
    for(int x = top; x>=0;x--)
        cout<<stk[x]<<endl;
    getch();
} //end of display()
```

Output of the program:

```
1- Pushing stack
2- Popping stack
3- Display stack
4- Exit
Enter your option [1-4] ? 2
Value 36 is removed
```

5.1.3.3 Complexity Analysis of Stack Pushing and Popping

The algorithm for "pushing an item into a stack with size 'n'" is an insertion operation. This algorithm does not use any loop. It directly inserts an item at the top of the stack. It requires the same time or constant time or $O(1)$ for push operation.

The algorithm for "popping an item from the stack with size 'n'" is a deletion operation. This algorithm also does not use any loop. It directly removes an item located at the top of the stack. It requires the same time or constant time or $O(1)$ for pop operation.

The time complexity and space complexity of pushing and popping algorithms of the stack are as follows:

Operation	Time Complexity		Space Complexity
	Average-Case	Worst-Case	Worst-Case
Pushing (Insertion)	$\Theta(1)$	$O(1)$	$O(n)$
Popping (Deletion)	$\Theta(1)$	$O(1)$	$O(n)$

5.1.4 Decimal to Binary Conversion using Stack

A decimal number can be converted into an equivalent binary number using a stack. Following figure shows the process of converting decimal number 26 into a binary number:

2	26	
2	13	0
2	6	1
2	3	0
2	1	1
2	0	1

In the above process, the decimal number 26 is divided by 2 in the first step. The remainder of this step is 0 and the quotient is 13. Then the quotient part of the first step is divided by 2 (i.e. 13 is divided by 2). The remainder of this step is 1 and the quotient is 6. Repeatedly quotient is divided by 2 until it becomes zero. At the end of the division process, the answer is received by collecting the remainders of all the steps in reverse order. Therefore, the decimal number into an equivalent binary number is 11010.

The stack data structure can be used to convert a decimal number into a binary number. The push operation of the stack is used to store the remainder of each step of the division operation. After the division process, the remainders of all the steps can be popped. In this way the remainders will be removed in reverse order.

For example, the decimal number 26 can be converted into a binary number using stack:

- 1- Divide 26 by 2. The quotient of this step is 13, and the remainder is 0. Push the remainder i.e. 0 onto the stack.

0

- 2- Divide 13 by 2. The quotient of this step is 6, and the remainder is 1. Push the remainder i.e. 1 onto the stack.

1
0

- 3- Divide 6 by 2. The quotient of this step is 3, and the remainder is 0. Push the remainder i.e. 0 onto the stack.

0
1
0

- 4- Divide 3 by 2. The quotient of this step is 1, and the remainder is 1. Push the remainder i.e. 1 onto the stack.

1
0
1
0

- 5- Divide 1 by 2. The quotient of this step is 0, and the remainder is 1. Push the remainder i.e. 1 onto the stack.

1
1
0
1
0

- 6- Pop all values from stack one by one and print each value in the same sequence i.e. 11010. It is the equivalent binary number of the decimal number 26.

Algorithm – Converting Decimal Number into Binary Number

Write an algorithm to convert a decimal number into a binary number.

- 1- START
- 2- Initialize an empty Stack
- 3- INPUT integer number in N
- 4- REPEAT Step-5 to Step-7 WHILE $N > 0$
- 5- Divide N by 2
- 6- Push the remainder part of step-5 onto the Stack
- 7- Store the Quotient part of step-5 back into N
- [End of Loop of step-4]
- 8- Pop all values of Stack one by one and print on the screen
- 9- END

Program Code 5.2

Write a program that converts a decimal integer value into binary and displays the result on the screen. For example, if the number is 3 then the result should be displayed as $(11)_2$. Use the stack technique to store the remainder of each step of division and pop the values from stack and display on the screen.

```
#include <iostream.h>
#include <conio.h>

class stack
{
    private:
        int top;
        int stk[15];
    public:
        stack(void) { top = -1; }
        // Prototypes of the functions of the class
        void bin(int x);
        void display(void);
};

void main(void)
{
    stack obj;
    int n;
    clrscr();
    cout<<"Enter an integer value : ";
    cin>>n;
    obj.bin(n);
    obj.display();
    getch();
} //end of main()

// member function to convert number into binary and push onto stack
void stack::bin(int x)
{
    int r;
    while(x>0)
    {
        r = x%2;
        top = top+1;
        stk[top] = r;
        x = x/2;
    }
} //end of bin()

// member function to pop values from stack and display in binary format
void stack::display(void)
{

```



```

cout<<"Value in binary =(";
for(int i = top; i>=0; i--)
    cout<<stk[i];
cout<<" )2";
} //end of display()

```

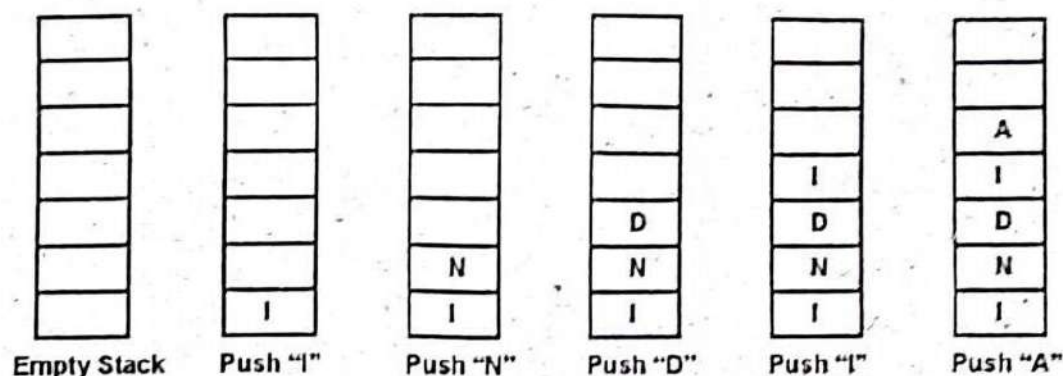
Output of the program:

Enter an integer value : 9

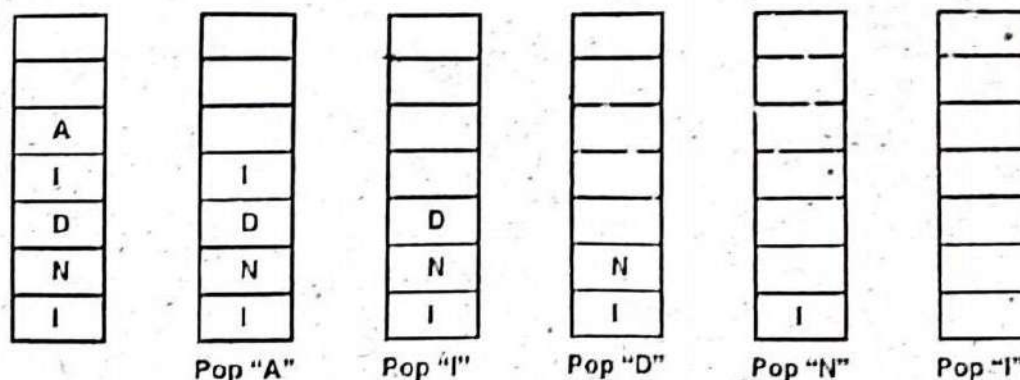
Value in binary = (1001)2

1.5 Reversing String using Stack

We can reverse a string very easily using stack. All characters of the given string (from left to right) are pushed onto the stack one by one until the end of the string. Suppose we push the characters of the string "INDIA" onto the stack. The sequence of pushing characters of string "INDIA" onto the stack is shown in the following diagrams:



When the characters of the string are popped from the stack, they come off in reverse order. Following figures shows the popping process of the string "INDIA":



After popping all characters from the stack, we will obtain the reverse string i.e. string "AIDNI" will be reversed as "INDIA".

Algorithm – Reversing String

Write an algorithm to reverse a given string using a stack.

- 1- START
- 2- Initialize an empty Stack
- 3- INPUT a string
- 4- Push all characters of the string one by one onto the stack
- 5- Pop all characters from the stack one by one and add them to the output string
- 6- Print the output string
- 7- END

Program Code 5.3

Write a program that reverses an input string using stack and displays the result on the screen. For example, if the input string is "ABCXYZ", it should be displayed as ZYXCBA.

```
#include <iostream.h>
#include <conio.h>
class stack
{
    private:
        int top;
        char stk[25];

    public:
        stack(void) { top = -1; }

        // Prototypes of the functions of the class
        void push_str(char x[]);
        void pop_str(void);
};

void main(void)
{
    stack obj;
    char str[25];
    clrscr();
    cout<<"Enter a string : ";
    cin>>str;
    obj.push_str(str);
    obj.pop_str();
    getch();
} //end of main()

// member function to push characters of string onto the stack
void stack::push_str(char x[])
{
    for(int i = 0; x[i]!='\0'; i++)
    {
        top++;
        stk[top] = x[i];
    }
}
```



```
    }  
} //end of push_str()  
  
// member function to pop characters of string from stack & display in reverse order.  
void stack::pop_str(void)  
{  
    cout<<"String in reverse order: ";  
    for(int i = top; i>=0; i--)  
        cout<<stk[i];  
} //end of pop_str()
```

Output of the program:

Enter a string : ABCDEF

String in reverse order : FEDCBA

5.2 APPLICATIONS OF STACK

As we have seen in previous examples, stacks are very efficient and easier data structures with respect to their implementation. They are used for maintaining the order of operations in complex programs. They are also used in management programs where the newest tasks must be executed first. For example, stacks are used in many application programs for undo and redo actions. In computer programming, stacks are mostly used in the following situations:

- Function Call
- Recursive Function
- Expression Conversion
- Expression Evaluation
- Games like "Towers of Hanoi"

5.3 FUNCTION CALL

A function is just a piece of code that can be referred to or called using its name. The computer stores the data of the function (such as arguments (parameters), local variables, return address, etc.) when a function is invoked (called for execution) and destroys them when the function returns to calling function. Similarly, when a function calls other functions, the caller function completes only after all the callee functions have completed their tasks i.e., the function that is invoked last completes first and the one invoked first completes last (If a function A calls function B then function A is the *caller* and function B is the *callee*). So the stack or LIFO (Last-In-First-Out) data structure can be used in this case.

In almost all programming languages, calling a function (recursive and non-recursive functions) involves the use of stack data structure. One function can call another function which can call the third one and so on to any depth until there is a final calculation or processing by the last called function. When there is a function call, all the important information that needs to be

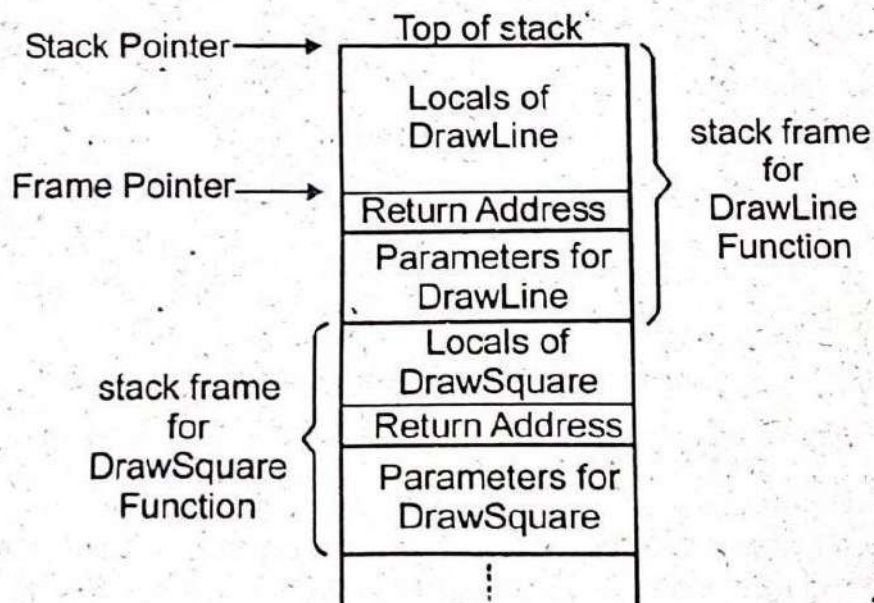
the return address which can be obtained from the program counter) are stored at the top of the stack. When a function calls another function, first its arguments, then the return address, and finally the space for local variables is pushed onto the stack. Since each function runs in its own "environment" or context, it becomes possible for a function to call itself - a technique known as *recursion*.

When the function has completed its task, the control goes back to the calling function. If it makes other function calls, it follows the same procedure. When the control goes back from the called function to the calling function, then all the previous registers' values of variables are restored.

Execution Stack or Call Stack

A data structure that is created and stores related information on the call of a function is called *Execution Stack*. It is also known as *Program Stack* or *Call Stack*.

The Execution Stack or Program Stack is used for many purposes but the main objective is to keep track of the point to which each active function should return control when it finishes executing. An active function is one that has been called but is yet to complete execution after which control should be handed back to the point of call to the function. Such activations of functions may be nested to any level. If, for example, a function *DrawSquare* calls another function *DrawLine* from different places, *DrawLine* must know where to return back when its execution completes. To accomplish this, the address following the call instruction, the return address, are pushed onto the call stack with each call.



Stack Frame or Activation Record

Execution or Program Stack is composed of stack frames (also called *activation records* or *activation frames*). Each stack frame consists of information related to call to a function. A stack frame is created whenever a function is called and destroyed when control goes back to the

calling function after completing the task. For example, if a function named `DrawLine` is called by a function `DrawSquare`, the Program Stack's position will be as shown in the following figure. In this figure, the first stack frame was created when the function `DrawSquare()` was called. However, the function `DrawSquare` called another function `DrawLine` before its completion. Thus, another stack frame (i.e. for function `DrawLine`) is created as shown in the figure. When the function `DrawLine` returns, its stack frame is popped. Similarly, when the function `DrawSquare` returns, its stack frame is also popped.

The stack frame at the top of the stack is for the currently executing function. The stack frame usually includes the following information:

- the parameter values passed to the function (if any).
- the return address back to the function's caller (e.g. in the `DrawLine` stack frame, an address into `DrawSquare`'s code).
- Memory space for the local variables of the function (if any).

Stack Pointer and Frame Pointer

The stack pointer is a memory address pointing to the top of the stack frame in the stack. The frame pointer is a memory address in the stack frame pointing to the location where control is returned on the execution of return statement in the related function.

When a function returns, the stack pointer is restored to the frame pointer, the value of the stack pointer just before the function was called. Each stack frame contains a stack pointer to the top of the frame. A frame pointer of a given call (invocation) of a function is a copy of the stack pointer as it was before the function was invoked. The locations of all other fields in the frame can be defined relative either to the top of the frame, as negative offsets of the stack pointer, or relative to the top of the frame below, as positive offsets of the frame pointer.

5.3.1 Processor's Stack Operation

There are two CPU registers that are important for the functioning of the stack. These registers hold information that is necessary when calling data residing in the memory. Names of these registers are ESP (Extended Stack Pointer) and EBP (Extended Base Pointer) in the 32 bits system. The ESP register serves as an indirect memory operand pointing to the top of the stack at any time.

In addition to the stack pointer which points to the top of the stack, there is Stack Frame Pointer (FP) which holds an address that points to a fixed location within a frame. Looking at the stack frame, local variables could be referenced by giving their offsets from ESP. However, as data are pushed onto the stack and popped off the stack, these offsets change, so the reference to the local variables does not remain consistent. Consequently, many compilers use another register, generally called Frame Pointer (FP), for referencing both local variables and parameters.

5.4 EXPRESSIONS

An arithmetic expression is made up of operands, arithmetic operators, and parentheses. The operands may be variables or constants. There are different types of expressions such as arithmetic expressions, relational expressions, and logical expressions. The type of expression depends upon the type of operators used to build the expression. For example, arithmetic operators are used to build arithmetic expressions. Following are some examples of arithmetic expressions:

$$A+B, A*B, A*(B-C)/2, (A+B)^2$$

The arithmetic operators +, -, *, and / are the same that are used in ordinary algebra. The operator ^ is used to compute the exponential. This operator is not available in C++. Instead, "pow()" function is used to calculate the exponential.

To solve an expression, each programming language has precedence and associativity (the order in which operators are evaluated by the compiler) of the operators. A stack is the most powerful data structure for solving or evaluating expressions.

Precedence

The order in which operations of an expression are evaluated is called the *order of precedence*. When an expression contains two or more different operators, the precedence of the operators controls the order in which the individual operators are evaluated.

An expression is always evaluated from left to right in the following order:

- ★ If the expression has parenthesis, the operations within parenthesis are evaluated first.
- ★ Exponential (^) is given the highest priority.
- ★ Multiplication (*) and division (/) have the next priority. They have equal priority and are evaluated from left to right.
- ★ Addition (+) and subtraction (-) have the lowest priority. They also have equal priority and are evaluated from left to right.

For example, the expression "x+y*z" is evaluated as "x+(y*z)" because the "*" operator has higher precedence than the binary "+" operator.

Associativity

When an expression contains two or more operators having the same precedence, associativity determines how operators of the same precedence are grouped in the absence of

parentheses and the order in which they execute. For example, in the expression " $a + b - c$ ", both $+$ and $-$ have the same precedence, then which part of the expression will be evaluated first, is determined by the associativity of these operators. Here, both $+$ and $-$ are left-associative, so the expression will be evaluated as " $(a + b) - c$ ".

Following table shows the precedence and associativity of arithmetic operators:

Operator	Precedence	Associativity
Exponentiation $^$	Highest	Right Associative
Multiplication ($*$) & Division ($/$)	Second Highest	Left Associative
Addition ($+$) & Subtraction ($-$)	Lowest	Left Associative

This table shows the default behavior of operators. The order of evaluation of an expression can be altered by using parenthesis. For example, in " $a + b * c$ ", the expression part " $b * c$ " will be evaluated first, because the multiplication operator " $*$ " has higher precedence over the addition operator " $+$ ". However, we can use parenthesis for " $a + b$ " to evaluate this part first such as $(a + b) * c$.

4.1 Notations used for Expressions

The method to write an arithmetic expression is known as a *notation*. In the process of evaluation of an expression, we need to represent it into a specific notation. There are three forms of notations that can be used to represent an arithmetic expression (depending on the placement of the operators & operands). These notations are *infix* notation, *prefix* notation, and *postfix* notation.

Infix Notation

The notation in which operators are written (placed) in between their operands is called *infix notation*. For example;

$$A + B$$

In this notation, operators are used in-between operands. In most arithmetic expressions, a binary arithmetic operator is placed between two operands such as $A * B$, A / B . Infix notation is the most general notation used for representing expressions.

Prefix Notation (Polish Notation)

The notation in which operators are written (placed) before their operands is called *prefix notation*. For example, the sum of A and B is written as:

$$+ A B$$

The prefix notation was introduced by the Polish mathematician Jan Lukasiewicz in 1920s. For this reason, *prefix* notation is also known as *Polish Notation*. The main property of Polish Notation is that the order in which operations are to be performed is

examined by the position of the operators and operands in the expression. Parentheses are not used in Polish Notation.

4.5) Postfix Notation or Reverse Polish Notation (RPN)

The notation in which operators are written (placed) after their operands is called *postfix notation*. This notation is also called *reverse polish notation (RPN)* or *suffix notation*. For example, the sum of A and B is written as:

$AB+$

Generally, postfix expression is free from operator precedence. That is why it is preferred in the computer system. The computer system uses a postfix form to represent any expression.

Following table shows expressions in infix notation and equivalent prefix and postfix notations:

Infix Notation	Prefix Notation (Polish Notation)	Postfix Notation (Reverse Polish Notation)
$A+B$	$+AB$	$AB+$
$A*B$	$*AB$	$AB*$
A/B	$/AB$	$AB/$
$A-B$	$-AB$	$AB-$

The computer system evaluates an expression given in infix notation by converting it into postfix notation. The stack data structure is used to perform this conversion.

5.5 EXPRESSION CONVERSION

We can convert one type of expression to an equivalent another type of expression like *infix to postfix*, *infix to prefix*, *postfix to prefix*, and vice versa. The stack is used to convert an expression from one type to another.

→ operator will go in stack
→ operand will go in output

5.5.1 Infix to Postfix Conversion

During the conversion process from infix to postfix, operands are directly added to the output, and operators are stored onto the stack. They are popped from the stack and added to the output according to their precedence.

In an *infix* expression, the arithmetic operators appear between two operands. Parentheses can be used in the *infix* expression to specify the order of the operations. For example: $a + (b * c)$

But in the *postfix* expression, parentheses are not used. During the conversion process parentheses are treated as operators. The left parenthesis indicates the beginning of a sub-expression and when encountered, it is pushed onto the stack. The right parenthesis is never

shed to the stack. When it is encountered, operators are popped from the stack until the left parenthesis is encountered but do not add it to the output. At this point, the sub-expression originally enclosed by the parentheses has been converted to *postfix* expression. So the parentheses are discarded and these are not added to the output i.e. *postfix* expressions. When the end of the expression is reached, all the operators from the stack are popped and added to the output.

Following table shows expressions in *infix* notation and equivalent *postfix* notations:

Infix Notation	Postfix Notation	Comment
$A + B$	$A B +$	Add A and B.
A / B	$A B /$	Divide A by B.
$(A + B) * C$	$A B + C *$	Add A and B, Multiply the result by C.
$A + (B * C)$	$A B C * +$	Multiply B by C, Add A, and the result.
$A * B + C / D$	$A B * C D / +$	Multiply A and B, Divide C by D, Add the results.
$A * (B + C) / D$	$A B C + * D /$	Add B and C, Multiply by A, Divide by D.
$A * (B + C / D)$	$A B C D / + *$	Divide C by D, Add B, Multiply by A.

Algorithm – Infix to Postfix Conversion

Write an algorithm to convert *infix* expression to *postfix* expression.

- 1- START
- 2- Initialize an empty Stack
- 3- INPUT infix expression and store it in variable EXP
- 4- $C = 0$
- 5- REPEAT Step-6 to 11 WHILE $EXP[C] \neq \text{NULL}$
[Read all the elements or symbols one by one from left to right in a given *infix* expression until end of the expression]
- 6- $CH = EXP[C]$
- 7- IF CH is an operand, then add it to the output.
- 8- IF CH is a left parenthesis '(', then push it onto the Stack.
- 9- IF CH is an operator, then;
 - i) If Stack is empty, then push read operator onto the Stack.
 - ii) If Stack is not empty, then check the precedence of the top operator of Stack with the read operator. If the precedence of the top operator of Stack is higher or the same than the read operator, then pop the operator from the Stack and add to the output. Repeat this step (step-ii) for the next top operators of Stack one by one. At the end of this step (i.e. step-ii), push the read operator onto the Stack.
[Please note that left parenthesis in the Stack is assumed to have lower precedence than that of all other operators]
- 10- If CH is right parenthesis ')', then pop all operators from the Stack until the respective left parenthesis is encountered and add each popped operator to the output. Do not add '(' to the output, simply discard it.

- 11- $C = C + 1$
[End of Step-5 loop]
- 12- When the end of the *infix* expression is reached, pop all operators from the stack and add each popped operator to the output in the same sequence.
- 13- END

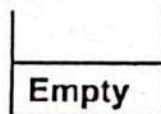
Example 1:

Convert the following *infix* notation into *postfix* notation using stack:

$$A * (B + C) - X / Y$$

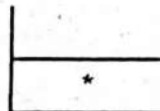
Following is the procedure to convert the *infix* notation into *postfix* notation using stack:

1. Scan *infix* expression from left to right. The first operand read is A, add it to output.



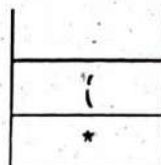
Output: A

2. Next the '*' operator is read. At this stage, the stack is empty, push '*' onto stack. Thus the status of the stack and the output will be:



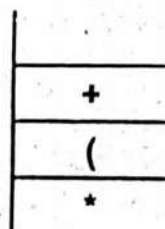
Output: A

3. The left parenthesis '(' is read. Since all operators have lower precedence than the left parenthesis, push it onto the stack. Then the operand 'B' is read, add it to the output. Thus the status of the stack and the output will be:



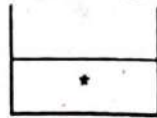
Output: A B

4. The next symbol read is the '+' operator, push it onto the stack (Because left parenthesis is placed at top of the stack; it cannot be popped from the stack until a right parenthesis is read.). Next the operand 'C' is read, add it to the output. Thus the status of the stack and the output will be:



Output: A B C

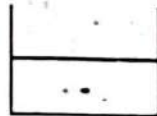
5. Now, the right parenthesis ')' is read. Pop all operators up to left parenthesis and add them to the output. Also, pop left parenthesis and discard it. Thus the status of the stack and the output will be:



Output: A B C +

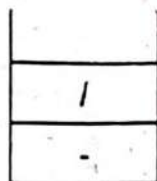
6. Next symbol read is the '-' operator. The stack has '*' operator which has higher precedence than the '-' operator. Pop the '*' operator from the stack, add it to the output and then push '-' onto the stack.

Next, the operand 'X' is read, add it to the output. Thus the status of the stack and the output will be:



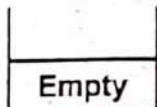
Output: A B C + * X

7. Now, the symbol read is '/' operator. The '-' operator in the stack has lower precedence than '/', therefore, push '/' onto the stack. The final symbol read is the operand 'Y', add it onto the output. Thus the status of the stack and the output will be:



Output: A B C + * X Y

8. Finally, the end of expression is reached. Pop all operators ('/' and '-') stored in the stack and add them to the output in the same sequence in which these are popped. The final output will be:



Output: A B C + * X Y / -

Example 2:

Convert the following *infix* expressions into the equivalent *postfix* expressions:

1. $(X-Y)*Z$
2. $X+(Y/Z)$
3. $M-N*O+(P*Q+R)/S$
4. $(3+5)*(9/((5-2)+1))$
5. $4*9/((3-5)+8)-7+2$
6. $A+B^5*C$
7. $X^Y/(4*Z)+9$

1- $(X-Y)*Z$

Following table shows the process of conversion of an *infix* expression into *postfix* expression:

Steps	Reading Symbol	Stack	Output
1	(	(	
2	X	(	X
3	-	(, -	X
4	Y	(, -	XY
5	)	Empty	XY -
6	*	*	XY - *
7	Z	*	XY - Z
8	End	Empty	XY - Z *

2- $X + (Y / Z)$

Following table shows the process of conversion of an *infix* expression into *post* expression:

Steps	Reading Symbol	Stack	Output
1	X	Empty	X
2	+	+	X
3	(	+, (	X
4	Y	+, (	XY
5	/	+, (, /	XY
6	Z	+, (, /	XYZ
7	)	+	XYZ /
8	End	Empty	XYZ / +

3- $M-N*O+(P*Q+R)/S$

Following table shows the process of conversion of an *infix* expression into *post* expression:

Steps	Reading Symbol	Stack	Output
1	M	Empty	M
2	-	-	M
3	N	-	MN
4	*	-, *	MN
5	O	-, *	MNO
6	+	+	MNO* -
7	(	+, (	MNO* -
8	P	+, (	MNO* - P

9	*	*, (, *	MNO* - P
10	Q	*, (, *	MNO* - PQ
11	+	*, (, +	MNO* - PQ*
12	R	*, (, +	MNO* - PQ*R
13	)	+	MNO* - PQ*R+
14	/	*, /	MNO* - PQ*R+
15	S	*, /	MNO* - PQ*R+S
16	End	Empty	MNO* - PQ*R+S/+

4- $(3+5)*(9/((5-2)+1))$

Following table shows the process of conversion of an *infix* expression into *posfix* expression:

Steps	Reading Symbol	Stack	Output
1	(	(	
2	3	(	3
3	+	(, +	3
4	5	(, +	35
5	)	Empty	35+
6	*	*	35+
7	(	*, (	35+
8	9	*, (	35+9
9	/	*, (/	35+9
10	(	*, (/,	35+9
11	(	*, (/,(	35+9
12	5	*, (/,(	35+95
13	-	*, (/,(,-	35+95
14	2	*, (/,(,-	35+952
15	)	*, (/,(	35+952-
16	+	*, (/,(+	35+952-
17	1	*, (/,(+	35+952-1
18	)	*, (/	35+952-1+
19	)	*	35+952-1+/*
20	End	Empty	35+952-1+/*

5- $4*9/((3-5)+8)-7+2$

Following table shows the process of conversion of an *infix* expression into *postfix* expression:

Steps	Reading Symbol	Stack	Output
1	4	Empty	4
2	*	*	4
3	9	*	4 9
4	/	/	4 9 *
5	(	/, (	4 9 *
6	(	/, (, (	4 9 *
7	3	/, (, (	4 9 * 3
8	-	/, (, (, -	4 9 * 3
9	5	/, (, (, -	4 9 * 3 5
10	)	/, (	4 9 * 3 5 -
11	+	/, (, +	4 9 * 3 5 -
12	8	/, (, +	4 9 * 3 5 - 8
13	)	/	4 9 * 3 5 - 8 +
14	-	-	4 9 * 3 5 - 8 + /
15	7	-	4 9 * 3 5 - 8 + / 7
16	+	+	4 9 * 3 5 - 8 + / 7 -
17	2	+	4 9 * 3 5 - 8 + / 7 - 2
18	End	Empty	4 9 * 3 5 - 8 + / 7 - 2 +

6- $A+B^5C$

Following table shows the process of conversion of an *infix* expression into *postfix* expression:

Steps	Reading Symbol	Stack	Output
1	A	Empty	A
2	+	+	A
3	B	+	A B
4	^	+, ^	A B
5	5	+, ^	A B 5
6	*	+, *	A B 5 ^
7	C	+, *	A B 5 ^ C
8	End	Empty	A B 5 ^ C * +

7- $X^Y/(4*Z)+9$

Following table shows the process of conversion of an *infix* expression into *postfix* expression:

Steps	Reading Symbol	Stack	Output
1	X	Empty	X
2	^	^	X
3	Y	^	XY
4	/	/	XY^
5	(	/(	XY^
6	4	/(	XY^4
7	*	/(,*	XY^4
8	Z	/(,*	XY^4Z
9	)	/	XY^4Z^
10	+	+	XY^4Z^/
11	9	+	XY^4Z^/9
12	End	Empty	XY^4Z^/9+

Program Code 5.4

// Program that converts infix expression into postfix expression.

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
#include <ctype.h>
```

```
int priority(char); // prototype of function to check precedence of operator
```

```
class infix_postfix
```

```
{
```

```
private:
```

```
char stack[20];
```

```
int top;
```

```
public:
```

```
infix_postfix() { top = -1;}
```

```
void scan(char []);
```

```
void push(char);
```

```
char pop(void);
```

```
};
```

```
void main(void)
```

```
{
```

```
infix_postfix obj;
```

```
clrscr();
```

```
char exp[100];
```

```
cout<<"Enter infix expression without spaces : ";
```

```
cin>>exp;
```

```
obj.scan(exp);
```

```
getch();
```

```
} // end of main()
```

```
// member function to read an expression and convert into postfix expression
```

```
void infix_postfix::scan(char str_exp[])
```

```
{
    char ch, output[80], temp;
    int c = 0;
    for (int i = 0; str_exp[i] != '\0'; i++)
    {
        ch = str_exp[i];

        // if ch is an operand, then add it to the output
        if (isalpha(ch) || isdigit(ch))
        {
            output[c] = ch;
            c++;
        }

        // if ch is left parenthesis '(' then push it onto the stack
        else if (ch == '(')
            push(ch);

        // pop all operators from stack if right parenthesis is read & add to output
        else if (ch == ')')
        {
            temp = pop();
            while (temp != '(')
            {
                output[c] = temp;
                c++;
                temp = pop();
            }
        }

        // if ch is an operator, then push or pop this operator onto or from
        // the stack on the basis of their precedence
        else if (ch == '^' || ch == '*' || ch == '/' || ch == '+' || ch == '-')
        {
            // push operator onto the stack if it is empty
            if (top == -1)
                push(ch);

            // if stack is not empty then repeatedly pop those operators
            // from stack that has higher or same precedence from

```



```

// read operator. After this push read operator onto st
else
{
    int op_ch;
    op_ch = priority(ch);
    while(priority(stack[top]) >= op_ch && top!=-1)
    {
        temp = pop();
        output[c] = temp;
        c++;
    }
    push(ch);
}
}
} // end of for loop that reads string of expression
// at the end of expression read, pop all operators from stack and add to output
for(int j = c; top>=0; j++)
    output[j] = pop();
output[j] = '\0';
cout<<"Postfix expression = "<<output;
} // end of scan()

// member function to push operator onto the stack
void infix_postfix::push(char x)
{
    top++;
    stack[top] = x;
} // end of push()

//member function to remove operators from the stack
char infix_postfix::pop(void)
{
    char r;
    r = stack[top];
    top--;
    return r;
} // end of pop()

//definition of function to check precedence of operator
int priority(char op)
{
    switch(op)
    {
        case '^': return 4; break;
        case '*':
    }
}

```

```

        case 'r': return 3; break;
        case '+':
        case '-': return 2; break;
        default: return 0;
    }
}

```

Output of the program:

Enter infix expression without spaces : A+B*C+(D*E+F)*G

Postfix expression = ABC*+DE*F+G*+

5.5.2 Infix to Prefix Conversion

As mentioned earlier, in *prefix* expression operators are written before their operands. For example, the *infix* expression "A+B" is equivalent to the *prefix* expression "+AB". The *infix* expression is converted to *prefix* expression in a similar way as an *infix* to *postfix* conversion. However, the *infix* expression is reversed, before converting to *prefix* expression. Similarly, right parenthesis ')' is pushed onto the stack, when left parenthesis '(' is encountered, then all operators from the stack are popped and added to the output. For example, *infix* expression A+B*C into reverse order is C*B+A.

Following table shows expressions in *infix* notations and equivalent *prefix* notations:

Infix Notation	Prefix Notation	Comment
A + B	+ A B	Add A and B.
A / B	/ A B	Divide A by B.
(A + B) * C	* + A B C	Add A and B, Multiply the result by C.
A + (B * C)	+ A * B C	Multiply B by C, Add A, and the result.
A * B + C / D	+ * A B / C D	Multiply A and B, Divide C by D, Add the results.
A * (B + C) / D	/ * A + B C D	Add B and C, Multiply by A, Divide by D.
A * (B + C / D)	* A + B / C D	Divide C by D, Add B, Multiply by A.

Algorithm – Infix to Prefix Conversion

Write an algorithm to convert infix expression to prefix expression.

- 1- START
- 2- Initialize an empty Stack
- 3- INPUT *infix* expression
- 4- Reverse the *infix* expression
- 5- Store the reverse *infix* expression into string variable EXP

- 6- $C = 0$
- 7- REPEAT Step-8 to 13 WHILE $EXP[C] \neq \text{NULL}$
[Read all the elements or symbols one by one from left to right in a given reverse *infix* expression until end of the expression]
- 8- $CH = EXP[C]$
- 9- IF CH is an operand, then add it to the output.
- 10- IF CH is right parenthesis ')', then push it onto the Stack.
- 11- IF CH is an operator, then;
 - i) If Stack is empty, then push the read operator onto the Stack.
 - ii) If Stack is not empty, then check the precedence of the top operator of Stack with the read operator. If the precedence of the top operator of Stack is higher or the same as the read operator, then pop the operator from the Stack and add to the output. Repeat this step (step-ii) for the next top operators of Stack one by one. At the end of this step (i.e. step-ii), push the read operator onto the Stack.
[Please note that right parenthesis in the Stack is assumed to have lower precedence than that of all other operators]
- 12- If CH is left parenthesis '(', then pop all operators from the Stack until the respective right parenthesis is encountered and add each popped operator to the output. Do not add ')' to the output, simply discard it.
- 13- $C = C + 1$
[End of Step-7 loop]
- 14- In the end, pop all elements of Stack and add to the output.
- 15- Reverse the output string/expression. The result will be in *prefix* expression of the given *infix* expression.
- 16- END

Example 1:

Convert the following *infix* expression into *prefix* expression using stack:

$$(L+Y)/T+M(A+B)/C$$

Following procedure is used to convert the *infix* notation into *prefix* notation using stack:

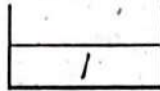
1. Convert the *infix* expression into reverse order. The expression $(L+Y)/T+M(A+B)/C$ into reverse order is as follows:

$$C/)B+A(M+T/)Y+L($$
2. Scan *infix* expression from left to right. The first operand read is C, add it to output.



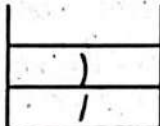
Output: C

3. Next the '/' operator is read. At this stage, the stack is empty, push '/' onto the stack. Thus the status of the stack and the output will be:



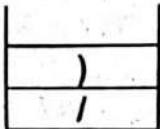
Output: C

4. Next the right parenthesis ')' is read. Push it onto the stack.



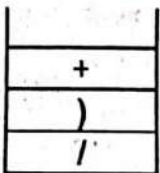
Output: C

5. Next, the operand read is B. Add it to the output.



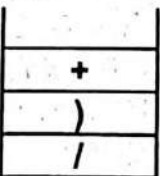
Output: C B

6. Next, the '+' operator is read. Push it onto the stack.



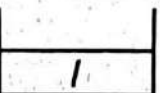
Output: C B

7. The next operand read is A. Add it to the output.



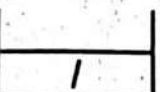
Output: C B A

8. Next, the left parenthesis '(' is read. Pop all operators up to right parenthesis ')' and add them to the output. Also, pop the right parenthesis and discard it. Thus the status of the stack and the output will be:



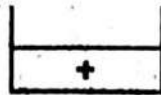
Output: C B A +

9. Next the operand read is M. Add it to the output.



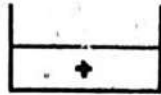
Output: C B A + M

10. Next operator read is '+' operator. The '+' operator has lower precedence than the operator '/' at the top of the stack, therefore, pop it from the stack and push '+' onto the stack.



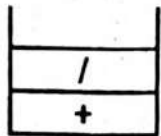
Output: C B A + M /

11. The next operand read is T. Add it to the output.



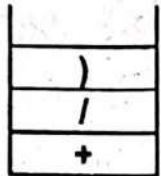
Output: C B A + M / T

12. Next operator read is '/' operator. The '/' operator has greater precedence than the operator '+' at the top of the stack, therefore, push it onto the stack.



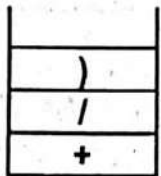
Output: C B A + M / T

13. Next the right parenthesis ')' is read. Push it onto the stack.



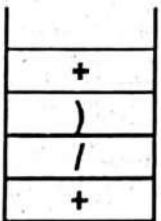
Output: C B A + M / T

14. The next operand read is Y. Add it to the output.



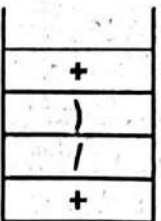
Output: C B A + M / T Y

15. Next operator read is '+' operator. Push it onto the stack.



Output: C B A + M / T Y

16. Next the operand read is L. Add it to the output.



Output: C B A + M / T Y L

17. Next the left parenthesis '(' is read. Pop all operators up to right parenthesis and add them to the output. Also, pop the right parenthesis and discard it. The status of the stack and the output will be:

/
+

Output: C B A + M / T Y L +

18. End of the reverse *infix* expression. Now pop all the elements from the stack and add to the output. Thus the status of the stack and the output will be:

Empty

Output: C B A + M / T Y L +

19. Reverse the output. The result will be the *prefix* expression of the given expression like this;

+ / + L Y T / M + A B C

Example 2:Convert the following *infix* expressions into *prefix* expressions:

1. $A + (B * C)$
2. $A * (B + C) / D$
3. $K * P / T * M + (A + B) / C$
4. $4 * 9 / ((3 - 5) + 8) - 7 + 2$
5. $A ^ (B + C) / (D - E) ^ 3$

A + (B * C)Convert the *infix* expression into reverse order. The expression $A + (B * C)$ into reverse order is as follows: $) C * B (+ A$ Following table shows the process to convert an *infix* into *prefix* expression:

Steps	Reading Symbol	Stack	Output
1	)	)	
2	C	)	C
3	*	), *	C
4	B	), *	C B
5	(	Empty	C B *
6	+	+	C B *
7	A	+	C B * A
8	End	Empty	C B * A +

Reverse the output obtained from the above table. The result will be the *prefix* expression of the given *infix* expression:

+ A * B C

$$A * (B + C) / D$$

Convert the *infix* expression into reverse order. The expression $A * (B + C) / D$ into reverse order is as follows:

$$D /) C + B (* A$$

Following table shows the process to convert an *infix* into *prefix* expression:

Steps	Reading Symbol	Stack	Output
1	D	Empty	D
2	/	/	D
3	)	/,)	D
4	C	/,)	D C
5	+	/,), +	D C
6	B	/,), +	D C B
7	(	/	D C B +
8	*	*	D C B + /
9	A	*	D C B + / A
10	End	Empty	D C B + / A *

Reverse the output obtained from the above table. The result will be the *prefix* expression of the given *infix* expression:

$$* A / + B C D$$

$$K * P / T * M + (A + B) / C$$

Convert the *infix* expression into reverse order. The expression $K * P / T * M + (A + B) / C$ into reverse order is as follows:

$$C /) B + A (+ M * T / P * K$$

Following table shows the process to convert an *infix* into *prefix* expression:

Steps	Reading Symbol	Stack	Output
1	C	Empty	C
2	/	/	C
3	)	/,)	C
4	B	/,)	C B
5	+	/,), +	C B
6	A	/,), +	C B A
7	(	/	C B A +
8	+	+	C B A + /
9	M	+	C B A + / M

10	*	+, *	CBA+ / M
11	T		CBA+ / MT
12	/	+, /	CBA+ / MT *
13	P	+, /	CBA+ / MT * P
14	*	+, *	CBA+ / MT * P /
15	K	+, *	CBA+ / MT * P / K
16	End	Empty	CBA+ / MT * P / K * +

Reverse the output obtained from the above table. The result will be the required *prefix* expression of the given *infix* expression:

$$+ * K / P * T M / + A B C$$

$$4 * 9 / ((3 - 5) + 8) - 7 + 2$$

Convert the *infix* expression into reverse order. The expression $4 * 9 / ((3 - 5) + 8) - 7 + 2$ into reverse order is as follows:

$$2 + 7 -) 8 +) 5 - 3 ((/ 9 * 4$$

Following table shows the process to convert an *infix* into *prefix* expression:

Steps	Reading Symbol	Stack	Output
1	2	Empty	2
2	+	+	2
3	7	+	2 7
4	-	-	2 7 +
5	)	-,)	2 7 +
6	8	-,)	2 7 + 8
7	+	-,), +	2 7 + 8
8	)	-,), +,)	2 7 + 8
9	5	-,), +,)	2 7 + 8 5
10	-	-,), +,), -	2 7 + 8 5
11	3	-,), +,), -	2 7 + 8 5 3
12	(	-,), +	2 7 + 8 5 3 -
13	(	-	2 7 + 8 5 3 - +
14	/	-, /	2 7 + 8 5 3 - +
15	9	-, /	2 7 + 8 5 3 - + 9
16	*	-, *	2 7 + 8 5 3 - + 9 /
17	4	-, *	2 7 + 8 5 3 - + 9 / 4
18	End	Empty	2 7 + 8 5 3 - + 9 / 4 * -

Reverse the output obtained from the above table. The result will be the *prefix* expression of the given *infix* expression:

$$- * 4 / 9 + - 3 5 8 + 7 2$$

5- $A ^ (B + C) / (D - E) ^ 3$

Convert the *infix* expression into reverse order. The expression $A ^ (B + C) / (D - E) ^ 3$ into reverse order is as follows:

$$3 ^) E - D (/) C + B (^ A$$

Following table shows the process to convert an *infix* into *prefix* expression:

Steps	Reading Symbol	Stack	Output
1	3	Empty	3
2	^	^	3
3	)	^,)	3
4	E	^,)	3 E
5	-	^,), -	3 E
6	D	^,), -	3 E D
7	(	^	3 E D -
8	/	/	3 E D - ^
9	)	/,)	3 E D - ^
10	C	/,)	3 E D - ^ C
11	+	/,), +	3 E D - ^ C
12	B	/,), +	3 E D - ^ C B
13	(	/	3 E D - ^ C B +
14	^	/, ^	3 E D - ^ C B +
15	A	/, ^, A	3 E D - ^ C B + A
16	End	Empty	3 E D - ^ C B + A ^ /

Reverse the output obtained from the above table. The result will be the *prefix* expression of the given *infix* expression:

$$/ ^ A + B C ^ - D E 3$$

Program Code 5.5

```
// Program that converts infix expression into prefix expression.
#include <iostream.h>
#include <conio.h>
#include <ctype.h>
#include <string.h>
```

```

char * reverse(char *); // function prototype for reversing the infix expression
int priority(char);      // prototype of function to check precedence of operator

class Infix_prefix
{
private:
    char stack[20];
    int top;
public:
    Infix_prefix() { top = -1; }
    void scan(char []);
    void push(char);
    char pop(void);
};

void main(void)
{
    Infix_prefix obj;
    clrscr();
    char exp[100], rev_exp[100];
    cout<<"Enter infix expression without spaces : ";
    cin>>exp;

    // convert infix expression in reverse order
    strcpy(rev_exp, reverse(exp));
    obj.scan(rev_exp);
    getch();
} // end of main()

// member function to read reverse infix expression and convert it into prefix
void Infix_prefix::scan(char str_exp[])
{
    char ch, output[100], rev_output[100], temp;
    int c = 0;
    for (int i = 0; str_exp[i] != '\0'; i++)
    {
        ch = str_exp[i];
        // if ch is an operand, then add it to the output
        if (isalpha(ch) || isdigit(ch))
        {
            output[c] = ch;
            c++;
        }
    }
}

```



```

        // if ch is right parenthesis ')' then push it onto the stack
    else if(ch==')')
        push(ch);

        // pop all operators from stack if left parenthesis is read and add to output
    else if(ch=='(')
    {
        temp = pop();
        while(temp!= ')')
        {
            output[c] = temp;
            c++;
            temp = pop();
        }
    }

    // if ch is an operator, then push or pop this operator onto or from
    // the stack on the basis of their precedence
    else if(ch=='^' || ch=='*' || ch=='/' || ch=='+' || ch=='-')
    {
        // push operator onto the stack if it is empty
        if(top == -1)
            push(ch);

        // if stack is not empty then repeatedly pop those operators
        // from stack that has higher or same precedence from
        // read operator. After this, push read operator onto stack
        else
        {
            int op_ch;
            op_ch = priority(ch);
            while(priority(stack[top]) >= op_ch && top!=-1)
            {
                temp = pop();
                output[c] = temp;
                c++;
            }
            push(ch);
        }
    }
} // end of for loop that reads the expression

//at the end of expression read, pop all operators from the stack & add to the output
for(int j = c; top>=0; j++)
    output[j] = pop();

```

```

        // append end of character at the end of output string
        output[j] = '\0';

        // reverse the output string and store it into rev_output string
        strcpy(rev_output, reverse(output));

        // print the output string which is in prefix form
        cout<<"Prefix expression = "<<rev_output;
    } // end of scan()

    // member function to push operator onto the stack
    void infix_prefix::push(char x)
    {
        top++;
        stack[top] = x;
    } // end of push()

    //member function to remove operators from the stack
    char infix_prefix::pop(void)
    {
        char r;
        r = stack[top];
        top--;
        return r;
    } // end of pop()

    // definition of reverse function to reverse the order of infix expression
    char * reverse(char * p_exp)
    {
        char temp_stk[100], temp[100], i, j, t = -1;
        // push characters of string onto the stack 'temp_stk'
        for(i = 0; p_exp[i]!='\0'; i++)
        {
            t++;
            temp_stk[t] = p_exp[i];
        }
        // pop characters of string from stack 'temp_stk' and store in 'temp' in reverse order
        for(i = t, j = 0; i>=0; i--, j++)
            temp[j] = temp_stk[i];
        temp[j] = '\0'; //append null sign at the end of temp
        return temp;
    } // end of reverse()

```



```
//definition of the function to check the precedence of an operator
int priority(char op)
{
    switch(op)
    {
        case '^': return 4; break;
        case '*':
        case '/': return 3; break;
        case '+':
        case '-': return 2; break;
        default: return 0;
    }
} // end of priority()
```

Output of the program:

Enter infix expression without spaces : A ^ (B + C) / (D - E) ^ 3
Prefix expression = / ^ A + B C ^ - D E 3

Postfix to Infix Conversion

Though *postfix* expressions can easily and efficiently be evaluated through computers, however, they can be difficult for humans to read. Complex expressions using standard *infix* notation are often more readable than the corresponding *postfix* expressions. Sometimes, expressions are stored or generated in *postfix*, and we would like to convert them to *infix* for the purpose of reading and editing.

A similar stack-based approach is used for converting *postfix* expression to *infix* expression as used for converting *infix* to *postfix*. The general algorithm will work the same way, however, the intermediate results are stored onto the stack. These intermediate results are called *sub-expressions*.

Algorithm – Postfix to Infix Conversion

Write an algorithm to convert *postfix* expression into *infix* expression.

- 1- START
- 2- Initialize an empty Stack
- 3- INPUT *postfix* expression into string variable EXP
- 4- C = 0
- 5- REPEAT Step-6 to 9 WHILE EXP[C] != NULL
[Read all the elements or symbols one by one from left to right of the given *postfix* expression until end of the expression]
- 6- CH = EXP[C]
- 7- IF CH is an operand, then push it onto the Stack.
- 8- IF CH is an operator, then;
 - i) Pop the top two operands or *infix sub-expressions* from the Stack.

- ii) Use parenthesis around the sub-expressions according to the parenthesis rule. For example, if the read operator has precedence greater than the operator used in any sub-expression, then write the sub-expression in parenthesis.
 - iii) Combine both sub-expressions with the read operator.
 - iv) Push the resulting expression onto the Stack.
- 9- $C = C + 1$
[End of Step-5 loop]
- 10- At end of the loop, pop the final infix expression from the Stack.
- 11- END

Example 1:

Convert the following *postfix* expression into *infix* expression using stack: -

$1\ 2\ *\ 3\ 4\ *\ +\ 5\ *$

Input	Stack	Comment
1	1	Push it onto the stack.
2	1, 2	Push it onto the stack.
*	1*2	Pop the top two items from the stack and combine them with the read operator. Push the resulting expression "1*2" onto the stack.
3	1*2, 3	Push it onto the stack.
4	1*2, 3, 4	Push it onto the stack.
*	1*2, 3*4	Pop the top two items from the stack and combine them with the read operator. Push the resulting expression "3*4" onto the stack.
+	1*2+3*4	Pop the top two sub-expressions (items) from the stack and combine them with the read operator. Push the resulting expression "1*2+3*4" onto the stack. Note that neither of the sub-expressions is placed in parentheses because + has lower precedence than *.
5	1*2+3*4, 5	Push it onto the stack.
*	(1*2+3*4)*5	Pop the top two sub-expressions (items) from the stack and combine them with the read operator. Push the resulting expression "(1*2+3*4)*5" onto the stack. Note that the left-hand sub-expression is placed in parentheses because * has higher precedence than +, which is used in it. Since there is the end of the expression, this is the final <i>infix</i> expression.

Example 2:

Convert the following *postfix* expression into *infix* expression using stack:

$A\ B\ C\ +\ *\ D\ /$

Input	Stack	Comment
A	A	Push it onto the stack.
B	A, B	Push it onto the stack.
C	A, B, C	Push it onto the stack.
+	A, B + C	Pop the top two items from the stack and combine them with the read operator. Push the resulting expression "B+C" onto the stack.
*	A*(B + C)	Pop the top two sub-expressions (items) from the stack and combine them with the read operator. Push the resulting expression "A*(B+C)" onto the stack. Note that the right-hand sub-expression is placed in parentheses because * has higher precedence than +, which is used in it.
D	A*(B + C), D	Push it onto the stack.
/	A*(B + C)/D	Pop the top two sub-expressions (items) from the stack and combine them with the read operator. Push the resulting expression "A*(B + C)/D" onto the stack. Since there is the end of the expression, this is the final <i>infix</i> expression.

Program Code 5.6

```
// Program that converts postfix expression into infix expression.
#include <iostream.h>
#include <conio.h>
#include <ctype.h>
#include <string.h>

// prototype of function to check the precedence of an operator
int priority(char);

// prototype of function to check operator in an expression of lowest priority
int lowest_priority_operator(char *);

// prototype of function to write an expression in brackets
char* brackets(char *);

class postfix_infix
{
private:
    char stack[25][100];
    int top;
public:
    postfix_infix() { top = -1; }
    void scan(char *);
```

```

        void push(char *);
        char * pop(void);
    };

void main(void)
{
    postfix_infix obj;
    clrscr();
    char exp[100];
    cout<<"Enter postfix expression without spaces : ";
    cin>>exp;
    obj.scan(exp);
    getch();
} // end of main()

// member function to read an expression
void postfix_infix::scan(char * str_exp)
{
    char ch, ch_str[5], op_str[5];
    for (int i = 0; str_exp[i]!='\0'; i++)
    {
        ch = str_exp[i];
        if (isalpha(ch) || isdigit(ch))
        {
            // convert single character into string then push onto the stack
            ch_str[0] = ch;
            ch_str[1] = '\0';
            push(ch_str); // push onto the stack
        }

        // if ch is an operator, then pop two operands or infix
        // sub-expressions from the stack, combine them with read operator
        // and push the result onto the stack
        else if (ch=='^' || ch=='*' || ch=='/' || ch=='+' || ch=='-')
        {
            char str1[50], str2[50];
            char result[100];
            int op, op1, op2;

            //pop first sub-expression from stack and copy it into str1
            strcpy(str1, pop());

            //pop 2nd sub-expression from stack and copy it into str2
            strcpy(str2, pop());

            // check the operator that has lowest priority in str1

```



```

op1 = lowest_priority_operator(str1);

    // check the operator that has lowest priority in str2
op2 = lowest_priority_operator(str2);

    // check priority (precedence) of read operator
op = priority(ch);

    // write parenthesis around str1 if precedence value of
    // read operator is higher than any operator in str1
if(op > op1 && op1!=0)
    strcpy(str1,brackets(str1));

    // write parenthesis around str2 if precedence value of
    // read operator is higher than any operator in str1
if(op > op2 && op2!=0)
    strcpy(str2,brackets(str2));

    // convert read operator into string and store to op_str
op_str[0] = ch;
op_str[1] = '\0';

    //combine str2, read operator and str1
strcpy(result, str2);    //copy str2 into result
strcat(result, op_str);  //append op_str with result
strcat(result, str1);    // append str1 with result

    // push result onto the stack
push(result);
}
} // end of for loop that reads the expression

    // print final result from the stack
cout<<"Infix expression : ";
cout<<pop()<<endl;
} // end of scan()

// member function to push operand or sub-expression onto the stack
void postfix_infix::push(char x[])
{
    top++;
    strcpy(stack[top], x);
} // end of push()

//member function to pop sub-expression from the stack

```

```

char * postfix_infix::pop(void)
{
    char res[80];
    strcpy(res, stack[top]);
    top--;
    return res;
} // end of pop()

//definition of the function to check operator that has the lowest priority in sub-expression
int lowest_priority_operator(char * pstr)
{
    int pri[10]= {0}, c=-1, min, i;

    // this loop checks the operators in the expression
    // and stores their priority numbers in an array pri
    for(i= 0; pstr[i]!='\0';i++)
        if(pstr[i]== '^' || pstr[i]== '*' || pstr[i]== '/' || pstr[i]== '+' || pstr[i]== '-')
        {
            c++;
            pri[c] = priority(pstr[i]);
        }

    // this loop checks the operator that has lowest priority value in expression
    min = pri[0];
    i = 1;
    while(i<=c)
    {
        if(min.> pri[i])
            min = pri[i];
        i++;
    }
    return min;
} //end of lowest_priority_operator()

// definition of function to write sub-expression in brackets
char * brackets(char * pstr)
{
    char temp[50];
    strcpy(temp, "("); // copy '(' in temp
    strcat(temp, pstr); //combine pstr with temp
    strcat(temp, ")"); // add ')' at the end of temp
    return temp;
} // end of brackets()

//definition of the function to check the precedence of an operator
int priority(char op)
{

```



```
switch(op)
{
    case '^': return 4; break;
    case '**':
    case '/': return 3; break;
    case '+':
    case '-': return 2; break;
    default: return 0;
}
} // end of priority()
```

Output of the program:

Enter postfix expression without spaces : ABC**DE*F+G**

Infix expression : A+B*C+(D*E+F)*G

5.4 Prefix to Infix Conversion

Prefix to *infix* conversion method is similar to *postfix* to *infix* conversion. However, the *prefix* expression is reversed before converting it into an *infix* expression. Prefix expression is converted into *infix* expression for the purpose of reading and editing.

A similar stack-based approach is used for converting *prefix* expression to *infix* expression as used for converting *postfix* to *infix*.

Algorithm – Prefix to Infix Conversion

Write an algorithm to convert *prefix* expression to *infix* expression.

- 1- START
- 2- Initialize an empty Stack
- 3- INPUT *prefix* expression
- 4- Reverse *prefix* expression and store it into variable EXP
- 5- C = 0
- 6- REPEAT Step-7 to 10 WHILE EXP[C] != NULL
[Read all the elements or symbols one by one from left to right in the given reverse *prefix* expression until end of the expression]
- 7- CH = EXP[C]
- 8- IF CH is an operand, then push it onto the Stack.
- 9- IF CH is an operator, then;
 - i) Pop top two operands or *infix* sub-expressions from the Stack.
 - ii) Use parenthesis around the sub-expression according to the parenthesis rule. For example, If the read operator has precedence greater than the operator used in any sub-expression, then write that SUB-expression in parenthesis.

- iii) Combine these two operands or sub-expressions with the read operator.
 - iv) Push the resulting expression onto the Stack.
- 10- $C = C + 1$
[End of Step-6 loop]
- 11- At the end of loop, pop final *infix* expression from the Stack.
- 12- END

Example 1:

Convert the following *prefix* expression into *infix* expression using stack:

$+ - 4 3 5$

Convert the *prefix* expression into reverse order. The prefix expression $+ - 4 3 5$ in reverse order is as follows:

$5 3 4 - +$

Following table shows the process to convert a *prefix* into *infix* expression:

Input	Stack.	Comment
5	5	Push it onto the stack.
3	5, 3	Push it onto the stack.
4	5, 3, 4	Push it onto the stack.
-	5, 4-3	Pop the top two operands (items) from the stack and combine them with the read operator. Push the resulting expression onto the stack.
+	5+4-3	Pop the top two operands or sub-expressions (items) from the stack and combine them with the read operator. Push the resulting expression onto the stack. End of the reverse <i>prefix</i> expression. Pop the result from the stack. It will be the equivalent <i>infix</i> expression of the given <i>prefix</i> expression.

Example 2:

Convert the following *prefix* expression into *infix* expression using stack:

$/ * A + B C D$

Convert the *prefix* expression into reverse order. The prefix expression $/ * A + B C D$ into reverse order is as follows:

$D C B + A * /$

Following table shows the process to convert a *prefix* into *infix* expression:

Input	Stack	Comment
D	D	Push it onto the stack.
C	D, C	Push it onto the stack.
B	D, C, B	Push it onto the stack.
+	D, B+C	Pop the top two operands (items) from the stack and combine them with the read operator. Push the resulting expression onto the stack.
A	D, B+C, A	Push it onto the stack.
*	D, A*(B+C)	Pop the top two sub-expressions (items) from the stack and combine them with the read operator. Push the resulting expression onto the stack. Note that the right-hand sub-expression is placed in parentheses because * has higher precedence than +, which is used in it.
/	A*(B+C)/D	Pop the top two sub-expressions (items) from the stack and combine them with the read operator. Push the resulting expression onto the stack. End of the reverse <i>prefix</i> expression. Pop the result from the stack. It will be the equivalent <i>infix</i> expression of the given <i>prefix</i> expression.



To convert *prefix* expression to *infix* expression, when we *pop* the top two operands, the first top operand comes on the left side of the read operator, while the second top operand comes on the right side of the read operator. It is because *prefix* expression is reversed before conversion to *infix* conversion e.g. if 9 is the first top operand, 3 is the second top operand and '/' is the read operator, then:

$$9 / 3 = 3$$

Program Code 5.7

```
// Program that converts prefix expression into infix expression.
#include <iostream.h>
#include <conio.h>
#include <ctype.h>
#include <string.h>

// function prototype for reversing the infix expression
char * reverse(char *);

// prototype of function to check the precedence of an operator
int priority(char);

// prototype of function to check operator in an expression of lowest priority
int lowest_priority_operator(char *);

// prototype of function to write an expression in brackets
```

```

char* brackets(char*);

class prefix_infix
{
    private:
        char stack[25][100];
        int top;
    public:
        prefix_infix() { top = -1;}
        void scan(char *);
        void push(char *);
        char * pop(void);
};

void main(void)
{
    prefix_infix obj;
    clrscr();
    char exp[100], rev_exp[100];
    cout<<"Enter prefix expression without spaces : ";
    cin>>exp;

    // convert prefix expression in reverse order
    strcpy(rev_exp, reverse(exp));
    obj.scan(rev_exp);
    getch();
} // end of main()

// member function to read an expression
void prefix_infix::scan(char * str_exp)
{
    char ch, ch_str[5], op_str[5];
    for (int i = 0; str_exp[i]!='\0'; i++)
    {
        ch = str_exp[i];
        if((isalpha(ch)) || isdigit(ch))
        {
            // convert single character into string then push onto the stack
            ch_str[0] = ch;
            ch_str[1] = '\0';
            push(ch_str); // push onto the stack
        }

        // if ch is an operator, then pop two operands or infix
        // sub-expressions from the stack, combine them with read operator
    }
}

```



```

        // and push the result onto the stack
    else if(ch=='^' || ch=='*' || ch=='/' || ch=='+' || ch=='-')
    {
        char str1[50], str2[50];
        char result[100];
        int op, op1, op2;

        //pop first sub-expression from stack and copy it into str1
        strcpy(str1, pop());

        //pop 2nd sub-expression from stack and copy it into str2
        strcpy(str2, pop());

        // check the operator that has lowest priority in str1
        op1 = lowest_priority_operator(str1);
        // check the operator that has lowest priority in str2
        op2 = lowest_priority_operator(str2);

        // check priority (precedence) of read operator
        op = priority(ch);

        // write parenthesis around str1 if precedence value of
        // read operator is higher than any operator in str1.
        if(op > op1 && op1!=0)
            strcpy(str1, brackets(str1));

        // write parenthesis around str2 if precedence value of
        // read operator is higher than any operator in str1.
        if(op > op2 && op2!=0)
            strcpy(str2, brackets(str2));

        // convert read operator into string and store to op_str
        op_str[0] = ch;
        op_str[1] = '\0';

        //combine str1, read operator, str2
        strcpy(result, str1); //copy str1 into result
        strcat(result, op_str); //append op_str with result
        strcat(result, str2); //append str2 with result
        // push result onto the stack
        push(result);
    }
} // end of for loop that reads the expression

// print final result from the stack
cout<<"Infix expression : ";

```

```

        cout<<pop()<<endl;
    } // end of scan()

    // member function to push operand or sub-expression onto the stack
    void prefix_infix::push(char x[])
    {
        top++;
        strcpy(stack[top], x);
    } // end of push()

    //member function to pop sub-expression from the stack
    char * prefix_infix::pop(void)
    {
        char res[80];
        strcpy(res, stack[top]);
        top--;
        return res;
    } // end of pop()

    // definition of reverse function to reverse the order of infix expression
    char * reverse(char * p_exp)
    {
        char temp_stk[100], temp[100], i, j, t = -1;
        // push characters of string onto the stack 'temp_stk'
        for(i = 0; p_exp[i]!='\0'; i++)
        {
            t++;
            temp_stk[t] = p_exp[i];
        }

        // pop characters of string from stack 'temp_stk' and store in 'temp' in reverse order
        for(i = t, j = 0; i>=0; i--, j++)
            temp[j] = temp_stk[i];
        temp[j] = '\0'; //append null sign at the end of temp
        return temp;
    } // end of reverse()

    //definition of function to check operator that has lowest priority in sub-expression
    int lowest_priority_operator(char * pstr)
    {
        int pri[10] = {0}, c = -1, min, i;
        // this loop checks the operators in the expression
        // and stores their priority numbers in an array pri
        for(i = 0; pstr[i]!='\0'; i++)

```



```

        if(pstr[i]== '^' || pstr[i]== '*' || pstr[i]== '/' || pstr[i]== '+' || pstr[i]== '-')
        {
            c++;
            pri[c] = priority(pstr[i]);
        }
        // this loop checks the operator that has lowest priority value in expression
    min = pri[0];
    i = 1;
    while(i<=c)
    {
        if(min > pri[i])
            min = pri[i];
        i++;
    }
    return min;
} // end of lowest_priority_operator()

// definition of function to write sub-expression in brackets
char * brackets(char * pstr)
{
    char temp[50];
    strcpy(temp, "("); // copy '(' in temp
    strcat(temp, pstr); // combine pstr with temp
    strcat(temp, ")"); // add ')' at the end of temp
    return temp;
} // end of brackets()

//definition of the function to check the precedence of an operator
int priority(char op)
{
    switch(op)
    {
        case '^': return 4; break;
        case '*':
        case '/': return 3; break;
        case '+':
        case '-': return 2; break;
        default: return 0;
    }
} // end of priority()

```

Output of the program:

Enter prefix expression without space : / * A + B C D

Infix expression : (A*(B+C))/D

5.6 EXPRESSION EVALUATION

The most frequent application of stack is the evaluation of arithmetic expressions. The expressions can be in the form of *infix*, *postfix*, or *prefix* notations. *Infix* notations are more difficult to evaluate by computers than other notations (such as *prefix* notation and *postfix* notation). Instead, the *infix* notations are first converted into either *postfix* or *prefix* notations and then evaluated.

Many compilers use a stack for parsing the syntax of expressions, program source code (i.e. Java, C++ or XML source code, etc.) before translating the source code of a program into object code (machine code). The word "parse" means "evaluation" or "analysis". The parser is used for this purpose. It is the component of a compiler or interpreter. It breaks data into smaller elements for easy translation into low-level language (machine language). A parser takes input in the form of a sequence of tokens or program instructions and usually builds a data structure in the form of a parse tree or an abstract syntax tree.

Sometimes, evaluation of an expression is also known as *parsing of expression*. Parsing of an arithmetic expression follows precedence and associativity of operations. Precedence and associativity determine the order of evaluation of an expression.

5.6.1 Evaluation of Infix Expression

Infix notation is the common arithmetic and logical formula notation, in which operators are written *infix*-style between the operands. *Infix* expression is easy for reading and writing. However, *infix* expression is difficult for computers to evaluate because of the additional work needed to decide precedence. But many programming languages use it due to its familiarity.

Infix expression is evaluated using two stacks; one for operators and another for operands.

Algorithm – Evaluation of Infix Expression

Write an algorithm to evaluate an *infix* expression.

- 1- START
- 2- Initialize empty Operand Stack and Operator Stack
- 3- Store *infix* expression into string variable EXP
- 4- C = 0
- 5- REPEAT Step-6 to 11 WHILE EXP[C] != NULL
[Read all the elements or symbols one by one from left to right in the given *infix* expression until end of the expression]
- 6- CH = EXP[C]

- 7- IF CH is an Operand, then push it onto the Operand Stack
- 8- IF CH is left parenthesis '(', then push it onto the Operator Stack
- 9- IF CH is an Operator, then;
 - If Operator Stack is empty, then push the read operator onto Operator Stack.
 - If Operator Stack is not empty, then;
WHILE Operator Stack is not empty, and the top operator on the Operator Stack has higher or equal precedence than the read Operator
 - i) POP top operator from the Operator Stack
 - ii) POP two operands from Operand Stack
 - iii) Perform operation specified by the popped operator
 - iv) PUSH the result onto the Operand Stack

[End of While Loop]

- PUSH read operator onto the Operator Stack
- 10- IF CH is right parenthesis ')', then:
WHILE left parenthesis '(' is encountered in Operator Stack
 - i) POP top operator from the Operator Stack
 - ii) POP two operands from Operand Stack
 - iii) Perform operation specified by the popped operator
 - iv) PUSH the result onto the Operand Stack

[End of While Loop under Step-10]

- v) POP left parenthesis '(' from the top of Operator Stack and discard it.
- 11- $C = C + 1$
[End of Step-5 Loop]
- 12- REPEAT Steps 13 to 15 Until Operator Stack is not empty.
- 13- POP top operator from the Operator Stack
- 14- POP two operands from Operand Stack and then perform the operation specified by the popped operator
- 15- PUSH the result onto the Operand Stack
[End of Step-12 Loop]
- 16- POP final result from the Operand Stack
- 17- END

Example 1:

Solve the following given *infix* expression. Verify your answer using a stack.

$$4+5*3-2$$

Solution:

$$\begin{aligned} 4+5*3-2 &= 4+15-2 \\ &= 19-2 \\ &= 17 \end{aligned}$$

Verification of the answer using Stack

The above *infix* expression can be verified using a stack. For the verification of this expression, two stacks have to be used; one for operators and the other for operands. Suppose the two stacks are empty. Read all the elements or symbols one by one of the *infix* expression from left to right until the end of the expression. It takes the following steps to evaluate:

Input	Operand Stack	Operator Stack	Operation	Comment
4	4			Push '4' onto the Operand Stack.
+	4	+		Push '+' onto the Operator Stack.
5	4, 5	+		Push '5' onto the Operand Stack.
*	4, 5	+, *		Push '*' onto the Operator Stack. Because the precedence of '*' is higher than the operator '+' on the top of Operator Stack.
3	4, 5, 3	+, *		Push '3' onto the Operand Stack.
-	4, 15	+	$5 * 3 = 15$	The operator "+" on the top of Operator Stack has higher precedence than the read operator '-'. Pop two operands 5 and 3 from Operand Stack and operator * from Operator Stack. Perform multiplication operation and push the result onto the Operand Stack.
	19	-	$4 + 15 = 19$	Because in the top of Operator Stack, '+' has precedence equal to read operator '-'. Pop two operands 4 and 15 from Operand Stack and operator '+' from Operator Stack. Perform the addition operation and push the result onto the Operand Stack, then push read operator i.e. '-' onto the Operator Stack.
2	19, 2	-		Push '2' onto the Operand Stack. End of the expression.
	17		$19 - 2 = 17$	Perform operations until Operator Stack is not empty. Pop two operands 19 and 2 from Operand Stack and operator - from Operator Stack. Perform subtraction operation and push the result onto the Operand Stack.
End	17			

The final result in the Operand Stack is 17. The calculated result is also 17. Therefore, the answer is verified using a stack.

Example 2:

Show the sequence of operations performed on a stack or stacks to solve the following given *infix* expression:

$$3*5+4/2$$

Solution:

$$\begin{aligned} 3*5+4/2 &= 15+4/2 \\ &= 15+2 \\ &= 17 \end{aligned}$$

Verification of the answer using Stack

The above *infix* expression can be verified using a stack. For the verification of this expression, two stacks have to be used; one for operators and the other for operands. The sequence of operations performed on stacks to solve this expression is shown in the following table:

Input	Operand Stack	Operator Stack	Operation	Comment
3	3			Push '3' onto the Operand Stack.
*	3	*		Push '*' onto the Operator Stack.
5	3, 5	*		Push '5' onto the Operand Stack.
+	15	+	$3*5 = 15$	The operator "*" on the top of Operator Stack has higher precedence than the read operator '+'. Pop two operands 5 and 3 from Operand Stack and operator * from Operator Stack. Perform multiplication operation and push the result onto the Operand Stack, then push read operator i.e. + onto the Operator Stack.
4	15, 4	+		Push '4' onto the Operand Stack.
/	15, 4	+, /		The operator '+' on the top of Operator Stack has lower precedence than the read operator '/'. So push operator '/' onto the Operator Stack.
2	15, 4, 2	+, /		Push '2' onto the Operand Stack. End of the expression.
	15, 2	+	$4/2 = 2$	Perform operations until Operator Stack is not empty. Pop two operands 4 and 2 from Operand Stack and operator '/' from top of Operator Stack. Perform operation division and push the result onto the Operand Stack.

	17		$15+2 = 17$	Pop two operands 15 and 2 from Operand Stack and operator '+' from Operator Stack. Perform the operation and push the result onto the Operand Stack.
End	17			

The final result in the Operand Stack is 17. The calculated result is also 17. Therefore, the answer is verified using a stack.

Example 3:

Solve the following given *infix* expression. Verify your answer using a stack.

$$9 + (5 * 3) / 3$$

Solution:

$$\begin{aligned}
 9 + (5 * 3) / 3 &= 9 + 15 / 3 \\
 &= 9 + 5 \\
 &= 14
 \end{aligned}$$

Verification of the answer using Stack

Following table shows the verification of the solution of the above expression using stack:

Input	Operand Stack	Operator Stack	Operation	Comment
9	9			Push '9' onto the Operand Stack.
+	9	+		Push '+' onto the Operator Stack.
(	9	+, (		Push '(' onto the Operator Stack.
5	9, 5	+, (		Push '5' onto the Operand Stack.
*	9, 5	+, (, *		Push '*' onto the Operator Stack.
3	9, 5, 3	+, (, *		Push '3' onto the Operand Stack.
)	9, 15	+	$5 * 3 = 15$	On right parenthesis ')', pop the operator on the top of Operator Stack. Pop two operands from Operand Stack. Perform the multiplication operation and push the result onto the Operand Stack. Repeat this step until left parenthesis '(' is encountered in Operator Stack. Pop '(' from Operator Stack and discard it.
/	9, 15	+, /		The operator '+' on the top of Operator Stack has lower precedence than the read operator '/'. So push operator '/' onto the Operator Stack.

3	9, 15, 3	+, /		Push onto the Operand Stack. End of expression.
	9, 5	+	$15/3 = 5$	Perform operations until Operator Stack is not empty. Pop two operands 15 and 3 from Operand Stack and operator '/' from Operator Stack. Perform division operation and push the result onto the Operand Stack.
	14		$9+5 = 14$	Pop two operands 9 and 5 from Operand Stack and operator '+' from Operator Stack. Perform the addition operation and push the result onto the Operand Stack.
End	14			

The final result in the Operand Stack is 14. The calculated result is also 14. Therefore, the answer is verified using a stack.

Example 4:

Solve the following given *infix* expression. Verify your answer using a stack.

$$2+3*(4-8*5)-1$$

Solution:

$$\begin{aligned}
 2+3*(4-8*5)-1 &= 2+3*(4-40)-1 \\
 &= 2+3*(-36)-1 \\
 &= 2-108-1 \\
 &= -106-1 \\
 &= -107
 \end{aligned}$$

Verification of the answer using Stack

Following table shows the verification of the above expression using stack:

Input	Operand Stack	Operator Stack	Operation	Comment
2	2			Push onto the Operand Stack.
+	2	+		Push onto the Operator Stack.
3	2, 3	+		Push onto the Operand Stack.
*	2, 3	+, *		The operator "+" on the top of Operator Stack has lower precedence than the read operator "*". So push, "*" onto the Operator Stack.
(	2, 3	+, *, (		Push onto the Operator Stack.
4	2, 3, 4	+, *, (		Push onto the Operand Stack.

-	2, 3, 4	+, *, (, -		Push onto the Operator Stack.
8	2, 3, 4, 8	+, *, (, -		Push onto the Operand Stack.
*	2, 3, 4, 8	+, *, (, -, *		The operator "-" on the top of Operator Stack has lower precedence than the read operator "*". So push "*" onto the Operator Stack.
5	2, 3, 4, 8, 5	+, *, (, -, *		Push onto the Operand Stack.
)	2, 3, 4, 40	+, *, (, -	$8 * 5 = 40$	On right parenthesis ')', pop the operator on the top of Operator Stack. Pop two operands from Operand Stack and perform the operation and push the result onto the Operand Stack. Repeat this step until left parenthesis '(' is encountered in Operator Stack. At the end of this step, pop '(' from Operator Stack and discard it.
	2, 3, -36	+, *, (	$4 - 40 = -36$	
	2, 3, -36	+, *		
-	2, -108	+	$3 * (-36) = -108$	The operator "*" on the top of Operator Stack has higher precedence than the read operator "-". Pop two operands from Operand Stack and operator "*" from Operator Stack. Perform multiplication operation and push the result onto the Operand Stack.
	-106	-	$2 + (-108)$ $2 - 108 = -106$	Because on the top of Operator Stack the operator '+' has precedence equal to read operator '-'. Pop two operands from Operand Stack and operator "+" from Operator Stack. Perform the addition operation and push the result onto the Operand Stack. At the end of this step, push read operator "-" onto the Operator Stack.
1	-106, 1	-		Push onto the Operand Stack. End of expression.
	-107		$-106 - 1 = -107$	Perform operations until Operator Stack is not empty. Pop two operands -106 and 1 from Operand Stack and operator "-" from Operator Stack. Perform subtraction operation and push the result onto the Operand Stack.
End	-107			

The final result in the Operand Stack is -107. The calculated result is also -107. Therefore, the answer is verified using a Stack.

Example 5:

Solve the following given *infix* expression. Verify your answer using a stack.

$$5*2-(9*5-3)+5$$

Solution:

$$\begin{aligned} 5*2-(9*5-3)+5 &= 10-(9*5-3)+5 \\ &= 10-(45-3)+5 \\ &= 10-42+5 \\ &= -32+5 \\ &= -27 \end{aligned}$$

Verification of the answer using a Stack

Following table shows the verification of the above *infix* expression using a stack:

Input	Operand Stack	Operator Stack	Operation	Comment
5	5			Push onto the Operand Stack.
*	5	*		Push onto the Operator Stack.
2	5, 2	*		Push onto the Operand Stack.
-	10	-	$5 * 2 = 10$	Precedence of the operator on the top of Operator Stack is higher than the read operator "-". Pop two operands from Operand Stack, and operator "*" from top of Operator Stack. Perform the operation and push the result onto the Operand Stack. After this, push the read operator onto the Operator Stack.
(	10	-, (		Push onto the Operator Stack.
9	10, 9	-, (		Push onto the Operand Stack.
*	10, 9	-, (, *		Push onto the Operator Stack.
5	10, 9, 5	-, (, *		Push onto the Operand Stack.
-	10, 45	-, (, -	$9 * 5 = 45$	Precedence of the operator on the top of Operator Stack is higher than the read operator "-". Pop two operands from Operand Stack, and operator "*" from top of Operator Stack. Perform the operation and push the result onto the Operand Stack. After this, push the read operator onto the Operator Stack.
3	10, 45, 3	-, (, -		Push onto the Operand Stack.
)	10, 42	-	$45 - 3 = 42$	On right parenthesis ')', pop the operator on the top of Operator Stack and pop two operands from Operand Stack. Perform

				the operation and push the result onto the Operand Stack. Repeat this step until left parenthesis '(' is encountered in Operator Stack. Pop '(' and discard it.
+	-32	+	10-42 = -32	Because the operator '-' on the top of Operator Stack has precedence equal to the read operator (+). Pop two operands from Operand Stack, and operator "-" from top of Operator Stack. Perform the operation and push the result onto the Operand Stack. After this, push the read operator onto the Operator Stack.
5	-32, 5	+		Push onto the Operand Stack. End of expression.
	-27		-32+5 = -27	Perform operations until Operator Stack is not empty. Pop two operands -32 and 5 from Operand Stack and operator "+" from Operator Stack. Perform the addition operation and push the result onto the Operand Stack.
End	-27			

The final result in the Operand Stack is -27. The calculated result is also -27. Therefore, the answer is verified using a stack.

Example 6:

Consider a stack. Illustrate what status of stacks (Operand and Operator stacks) will be after performing each of the following operations:

- Stack is initially empty
- 2, 3 and 7 are pushed
- Addition operator is pushed
- 6 is pushed
- Multiplication * operator is pushed

Stacks are initially empty. Following table shows the status of stacks after performing the above-given operations:

Operation	Operand Stack	Operator Stack	Arithmetic Operation	Comment
2, 3, and 7 are pushed	2, 3, 7			Push, onto the Operand Stack one after the other.
+	2, 3, 7	+		Push onto the Operator Stack.
6 is pushed	2, 3, 7, 6	+		Push onto the Operand Stack.

* is pushed	2, 3, 7, 6	+, *		The operator "+" on the top of Operator Stack has lower precedence than the read operator "*". So push the operator "*" onto the Operator Stack.
	2, 3, 42	+	7*6 = 42	Perform operations until Operator Stack is not empty. Pop two operands from Operand Stack and top operator "*" from Operator Stack. Perform the operation and push the result onto the Operand Stack.
	2, 45		3+42 = 45	Pop two operands from Operand Stack and operator "+" from Operator Stack. Perform the operation and push the result onto the Operand Stack.

The operand stack has two values 2 and 45 after performing the above-mentioned operation.

Program Code 5.8

// Program that evaluates infix expression.

```
#include <iostream.h>
#include <conio.h>
#include <math.h>
#include <ctype.h>
```

```
// prototype of function to convert a character digit to a numeric value
int character_to_value(char);
```

```
// prototype of function to check precedence of operator
int priority(char);
```

```
// prototype of function to perform arithmetic operation
int operation(int, char, int);
```

```
class infix
{
```

```
    private:
```

```
        int stack1[50];           // stack for operands
        char stack2[50];          // stack for operators
        int top1;
        int top2;
```

```
    public:
```

```

        infix()
        {
            top1 = -1;
            top2 = -1;
        }
        void scan(char []);           // function prototype to read expression
        void push1(int);              // function prototype to push operands
        void push2(char);             // function prototype to push operators
        int pop1(void);               // function prototype to pop operand
        char pop2(void);              // function prototype to pop operator
    };

    void main(void)
    {
        infix obj;
        clrscr();
        char exp[100];
        cout<<"Enter infix expression without spaces : ";
        cin>>exp;
        obj.scan(exp);
        getch();
    } // end of main()

    // member function to read an infix expression
    void infix::scan(char str_exp[])
    {
        char ch, op;
        int temp_val, v1, v2, res;
        for (int i = 0; str_exp[i]!='\0'; i++)
        {
            ch = str_exp[i];
            // if ch is digit (operand), convert it to numeric value and then
            // push onto the stack1 (stack for operands)
            if(isdigit(ch))
            {
                temp_val = character_to_value(ch);
                push1(temp_val);
            }

            // if ch is an alphabet (operand), input value for it and then
            // push onto the stack1 (stack for operands)
            else if(isalpha(ch))
            {
                cout<<"Enter value for "<<ch<<" : ";
                cin>>temp_val;
                push1(temp_val);
            }
        }
    }

```



```

        // if ch is left parenthesis, then push it onto the stack2
    else if( ch == '(' )
        push2(ch);

        // if ch is an operator
    else if(ch=='^' || ch=='*' || ch=='/' || ch=='+' || ch=='-')
    {
        // push operator onto the stack2 if it is empty
        if(top2==-1)
            push2(ch);

        // if stack2 is not empty then;
        else
        {
            int op_val;

            //check precedence of operator and then perform
            // action on the basis of precedence value of ch
            op_val = priority(ch);
            while(priority(stack2[top2]) >= op_val && top2!= -1)
            {
                op = pop2();
                v1 = pop1();
                v2 = pop1();
                res = operation(v2,op,v1);
                push1(res);
            }
            push2(ch);
        }
    }

    // if ch is right parenthesis
    else if(ch==')')
    {
        op = pop2();
        while(op!= '(')
        {
            v1 = pop1();
            v2 = pop1();
            res = operation(v2, op, v1);
            push1(res);
            op = pop2();
        }
    }
} // end of for loop that reads the expression

// at the end of expression read, pop all operators from stack2, values
// from stack1, and perform operation
while(top2>=0)

```

```

    {
        op = pop2();
        v1 = pop1();
        v2 = pop1();
        res = operation(v2, op, v1);
        push1(res);
    }

    // display the final evaluated result of expression from top of stack1
    cout<<"Result is : "<<stack1[top1];
} // end of scan()

// member function to push operand onto the stack1
void infix::push1(int val)
{
    top1++;
    stack1[top1] = val;
} // end of push1()

// member function to push operator onto the stack2
void infix::push2(char op)
{
    top2++; stack2[top2] = op;
} // end of push2()

//member function to remove operand from stack1
int infix::pop1(void)
{
    char r;
    r = stack1[top1];
    top1--;
    return r;
} // end of pop1()

//member function to remove operator from the stack2
char infix::pop2(void)
{
    char r;
    r = stack2[top2];
    top2--;
    return r;
} // end of pop2()

//definition of function to check precedence of operator
int priority(char op)
{
    switch(op)
    {

```



```
        case '^':      return 4; break;
        case '*':
        case '/':      return 3; break;
        case '+':
        case '-':      return 2; break;
        default:       return 0;
    }
} // end of priority()

//definition of function to perform arithmetic operation
int operation(int x, char op, int y)
{
    int res;
    switch(op)
    {
        case '*':      res = x * y; break;
        case '/':      res = x / y; break;
        case '+':      res = x + y; break;
        case '-':      res = x - y; break;
        case '^':      res = pow(x, y);
    }
    return res;
} // end of operation()

//definition of function to convert character digit into numeric value
int character_to_value(char x)
{
    int val;
    switch(x)
    {
        case '0':      val = 0; break;
        case '1':      val = 1; break;
        case '2':      val = 2; break;
        case '3':      val = 3; break;
        case '4':      val = 4; break;
        case '5':      val = 5; break;
        case '6':      val = 6; break;
        case '7':      val = 7; break;
        case '8':      val = 8; break;
        case '9':      val = 9; break;
    }
    return val;
} // end of character_to_value()
```

Output of the program:

Enter infix expression without spaces : 9+2^3*2

Result is : 25

5.6.2 Evaluation of Fully Parenthesized Infix Expression

In an *infix* arithmetic expression, operators are placed between two operands such as " $2+3$ or $1+(2+3) * (4*5)$ ". A fully parenthesized *infix* arithmetic expression is an *infix* arithmetic expression where every operator and its arguments are contained in parentheses such as;

$(2+3)$ or $(1+((2+3)*(4*5)))$

In the evaluation of a fully parenthesized expression, operator precedence and associativity are not involved. Only the left parenthesis '(' and right parenthesis ')' are involved. A fully parenthesized expression can be evaluated using two stacks such as operand stack and operator stack. Following algorithm describes the fully parenthesized expression:

Algorithm – Evaluation of Fully Parenthesized Infix Expression

Write an algorithm to evaluate a fully parenthesized *infix* expression.

- 1- START
- 2- Initialize empty Operand Stack and Operator Stack
- 3- Store *infix* expression into string variable EXP
- 4- $C = 0$
- 5- REPEAT Step-6 to 11 WHILE $EXP[C] \neq \text{NULL}$
 [Read *infix* expression from left to right until the end of the expression]
- 6- $CH = EXP[C]$
- 7- IF CH is '(', then push it onto the Operator Stack
- 8- IF CH is an operand, then push it onto the Operand Stack
- 9- IF CH is operator, then push it onto the Operator Stack
- 10- IF CH is ')', then;
 - i) POP operator from the top of Operator Stack
 - ii) POP two operands from Operand Stack
 - iii) Perform operation specified by the popped operator
 - iv) PUSH the result onto the Operand Stack
 - v) POP '(' from the top of Operator Stack and discard it.
- END IF
- 11- $C = C + 1$
 [End of Step-5 Loop]
- 12- POP final result from the Operand Stack.
- 13- END

Example 1:

Solve the following given *infix* expression. Verify your answer using stack.

$$((3*4) + (10/5))$$

Solution:

$$\begin{aligned} ((3*4) + (10/5)) &= (12 + 2) \\ &= 14 \end{aligned}$$

Verification of the answer using Stack

Following table shows the verification of the above fully parenthesized *infix* expression using stack:

Input	Operand Stack	Operator Stack	Operation	Comment
(		(		Push onto the Operator Stack.
(		(, (		Push onto the Operator Stack.
3	3	(, (		Push onto the Operand Stack.
*	3	(, (, *		Push onto the Operator Stack.
4	3, 4	(, (, *		Push onto the Operand Stack.
)	12	(	$3*4 = 12$	On right parenthesis ')', pop the operator on the top of Operator Stack. Pop two operands from Operand Stack, perform the operation, and then push the result onto the Operand Stack. Pop '(' from the Operator Stack and discard it.
+	12	(, +		Push onto the Operator Stack.
(	12	(, +, (		Push onto the Operator Stack.
10	12, 10	(, +, (		Push onto the Operand Stack.
/	12, 10	(, +, (, /		Push onto the Operator Stack.
5	12, 10, 5	(, +, (, /		Push onto the Operand Stack.
)	12, 2	(, +	$10/5 = 2$	On right parenthesis ')', pop the operator on the top of Operator Stack. Pop two operands from Operand Stack, perform the operation, and then push the result onto the Operand Stack. Pop '(' from the Operator Stack and discard it.
)	14		$12 + 2 = 14$	On right parenthesis ')', pop the operator on the top of Operator Stack. Pop two operands from Operand Stack, perform the operation, and push the result onto the Operand Stack. Pop '(' from the Operator Stack and discard it.
End	14			

The final result in the Operand Stack is 14. The calculated result is also 14. Therefore, the answer is verified using a Stack.

Example 2:

Solve the following given expression. Verify your answer using a stack.

$$(1 + ((2 + 3) * (4 * 5)))$$

Solution:

$$\begin{aligned} (1 + ((2 + 3) * (4 * 5))) &= (1 + ((2 + 3) * (4 * 5))) \\ &= (1 + (5 * (4 * 5))) \\ &= (1 + (5 * 20)) \\ &= (1 + 100) \\ &= 101 \end{aligned}$$

Verification of the answer using Stack

Following table shows the verification of the above fully parenthesized *infix* expression using stack:

Input	Operand Stack	Operator Stack	Operation	Comment
(		(		Push onto the Operator Stack.
1	1	(		Push onto the Operand Stack.
+	1	(, +		Push onto the Operator Stack.
(	1	(, +, (		Push onto the Operator Stack.
(	1	(, +, (, (		Push onto the Operator Stack.
2	1, 2	(, +, (, (		Push onto the Operand Stack.
+	1, 2	(, +, (, (, +		Push onto the Operator Stack.
3	1, 2, 3	(, +, (, (, +		Push onto the Operand Stack.
)	1, 5	(, +, (	2 + 3 = 5	On right parenthesis ')', pop the operator on the top of Operator Stack. Pop two operands from Operand Stack, perform the operation, and push the result onto the Operand Stack. Pop '(' from the Operator Stack and discard it.
*	1, 5	(, +, (, *		Push onto the Operator Stack.
(	1, 5	(, +, (, *, (		Push onto the Operator Stack.
4	1, 5, 4	(, +, (, *, (		Push onto the Operand Stack.
*	1, 5, 4	(, +, (, *, (, *		Push onto the Operator Stack.
5	1, 5, 4, 5	(, +, (, *, (, *		Push onto the Operand Stack.
				Push onto the Operator Stack.

)	1, 5, 20	(, +, (, *	$4 * 5 = 20$	On right parenthesis ')', pop the operator on the top of Operator Stack. Pop two operands from Operand Stack, perform the operation, and then push the result onto the Operand Stack. Pop '(' from the Operator Stack and discard it.
)	1, 100	(, +	$5 * 20 = 100$	On right parenthesis ')', pop the operator on the top of Operator Stack. Pop two operands from Operand Stack, perform the operation, and then push the result onto the Operand Stack. Pop '(' from the Operator Stack and discard it.
)	101		$1 + 100 = 101$	On right parenthesis ')', pop the operator on the top of Operator Stack. Pop two operands from Operand Stack, perform the operation, and then push the result onto the Operand Stack. Pop '(' from the Operator Stack and discard it.
End	101			

The final result in the Operand Stack is 101. The calculated result is also 101. Therefore, the answer is verified using Stack.

Program Code 5.9

// Program that evaluates infix expression.

```
#include <iostream.h>
#include <conio.h>
#include <math.h>
#include <ctype.h>
```

```
// prototype of function to convert a character digit to a numeric digit
int character_to_value(char);
```

```
// prototype of function to perform arithmetic operation
int operation(int, char, int);
```

```
class infix
{
```

```
    private:
```

```
        int stack1[50];
```

```
        // stack for operands
```

```
        char stack2[50];
```

```
        // stack for operators
```

```
        int top1;
```

```
        int top2;
```

```
    public:
```

```
        infix()
```

```

        {
            top1 = -1;
            top2 = -1;
        }
        void scan(char []);           // function prototype to read expression
        void push1(int);             // function prototype to push operands
        void push2(char);           // function prototype to push operators
        int pop1(void);              // function prototype to pop operand
        char pop2(void);            // function prototype to pop operator
    };

void main(void)
{
    infix obj;
    clrscr();
    char exp[100];
    cout<<"Enter infix expression without spaces : ";
    cin>>exp;
    obj.scan(exp);
    getch();
} // end of main()

// member function to read an infix expression
void infix::scan(char str_exp[])
{
    char ch, op;
    int temp_val, v1, v2, res;
    for (int i = 0; str_exp[i]!='\0'; i++)
    {
        ch = str_exp[i];

        // if ch is digit (operand), convert it to numeric digit and then
        // push onto the stack1 (stack for operands)
        if(isdigit(ch))
        {
            temp_val = character_to_value(ch);
            push1(temp_val);
        }

        // if ch is an alphabet (operand), input value for it and then
        // push onto the stack1 (stack for operands)
        else if(isalpha(ch))
        {
            cout<<"Enter value for "<<ch<<" : ";
            cin>>temp_val;
            push1(temp_val);
        }

        // if ch is left parenthesis, then push it onto the stack2
    }
}

```



```
        else if(ch=='(')
            push2(ch);

            // if ch is an operator , then push it onto the stack2
        else if(ch=='^' || ch=='*' || ch=='/' || ch=='+' || ch=='-')
            push2(ch);

            // if ch is right parenthesis
        else if(ch==')')
        {
            op = pop2();
            while(op!='(')
            {
                v1 = pop1();
                v2 = pop1();
                res = operation(v2, op, v1);
                push1(res);
                op = pop2();
            }
        }
    } //end of for loop that reads the expression

    // display the final evaluated result of expression from top of stack1
    cout<<"Result is : "<<stack1[top1];
} //end of scan()

// member function to push operand onto the stack1
void infix::push1(int val)
{
    top1++;
    stack1[top1] = val;
} //end of push1()

// member function to push operator onto the stack2
void infix::push2(char op)
{
    top2++;
    stack2[top2] = op;
} //end of push2()

//member function to remove operand from stack1
int infix::pop1(void)
{
    char r;
    r = stack1[top1];
    top1--;
    return r;
} // end of pop1()
```

```

//member function to remove operator from the stack2
char infix::pop2(void)
{
    char r;
    r = stack2[top2];
    top2--;
    return r;
} // end of pop2()

```

```

//definition of function to perform arithmetic operation
int operation(int x, char op, int y)
{
    int res;
    switch(op)
    {
        case '*':    res = x * y; break;
        case '/':    res = x / y; break;
        case '+':    res = x + y; break;
        case '-':    res = x - y; break;
        case '^':    res = pow(x, y);
    }
    return res;
} // end of operation()

```

```

//definition of function to convert character digit into numeric value
int character_to_value(char x)
{
    int val;
    switch(x)
    {
        case '0':    val = 0; break;
        case '1':    val = 1; break;
        case '2':    val = 2; break;
        case '3':    val = 3; break;
        case '4':    val = 4; break;
        case '5':    val = 5; break;
        case '6':    val = 6; break;
        case '7':    val = 7; break;
        case '8':    val = 8; break;
        case '9':    val = 9; break;
    }
    return val;
} // end of character_to_value()

```

Output of the program:

Enter infix expression without spaces : ((9+2)+(3^2))
 Result is : 20

5.6.3 Evaluation of Postfix Expression

The expressions written in *postfix* form are evaluated faster than *infix* notation. Parentheses are not used in *postfix* expression. Generally, *postfix* expressions are free from operator precedence that's why they are preferred in computer systems. A computer system uses a *postfix* form to represent an expression. A *postfix* expression can easily be evaluated using the stack data structure.

Algorithm – Evaluation of Postfix Expression

Write an algorithm to evaluate a *postfix* expression.

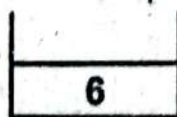
- 1- START
- 2- Initialize an empty Stack
- 3- INPUT *postfix* expression into string variable EXP
- 4- C = 0
- 5- REPEAT Step-6 to 9 WHILE EXP[C] != NULL
[Read the postfix expression from left to right until end of the expression]
- 6- CH = EXP[C]
- 7- IF CH is an Operand, then push operand onto the Stack
- 8- IF CH is an Operator, then:
 - i) POP top two operands of the Stack.
 - ii) Perform arithmetic operation on the operands (second top operand, operator, first top operand).
 - iii) PUSH the computed result onto the Stack.
- 9- C = C + 1
[End of Step-5 Loop]
- 10- POP computed result from the Stack.
- 11- PRINT the computed result.
- 12- END

Example 1:

Evaluate *postfix* expression AB/C-DE*+ when A = 6, B = 2, C = 1, D=3 & E=4.

The total number of items in the expression "AB/C -DE*+" are 9. So the loop will be executed 9 times.

1. In the first iteration, the value of A is pushed onto the Stack. The status of Stack will be:



2. In the second iteration, the value of B is pushed onto the Stack. The status of Stack will be:

2
6

3. In the third iteration, the operator '/' is encountered. So the two values 2 and 6 are popped (removed) from Stack and division operation is performed, i.e. $6/2 = 3$. The computed result is pushed back onto the Stack. The status of Stack will be:

3

4. In the fourth iteration, the value of C is pushed onto Stack. The status of Stack will be:

1
3

5. In the fifth iteration, the operator '-' is encountered. The two top values from the Stack are popped and arithmetic operation is performed, i.e. $3 - 1 = 2$. The result is pushed back onto the Stack. The status of Stack will be:

2

6. In the sixth iteration, the value of D is pushed onto the Stack. The status of Stack will be:

3
2

7. In the seventh iteration, the value of E is pushed onto the Stack. The status of Stack will be:

4
3
2

8. In the eighth iteration, multiplication operator '*' is encountered, thus the top two values 4 and 3 from Stack are popped. Multiplication operation is performed and calculated result 12 is pushed onto the Stack. The status of Stack will be:

12
2

9. In the ninth iteration, the operator '+' is encountered. Two values 12 and 2 from Stack are popped. The addition of these numbers is performed and the result 14 is pushed back onto the Stack. The top value in the Stack is 14. It is the final value after evaluating the expression:

14

Example 2:

Evaluate arithmetic expression $3*4+5$ using *postfix* notation.

The arithmetic expression $3*4+5$ in *postfix* notation is " $3\ 4\ *\ 5\ +$ ". Suppose the Stack is empty. Scan the *postfix* expression from left to right until the end of the expression. It takes the following steps to evaluate:

Input	Stack	Operation	Comment
3	3		Push onto the Stack.
4	3, 4		Push onto the Stack.
*	12	$3*4 = 12$	Pop values 4 and 3 from Stack. Perform multiplication operation $3*4$ and push result 12 onto the Stack.
5	12, 5		Push onto the Stack.
+	17	$12+5 = 17$	Pop values 5 and 12 from Stack. Perform the addition operation $12+5$ and push result 17 onto the Stack. This is the final result.

The top of the stack shows the final result of the expression.

Example 3:

Evaluate arithmetic expression $2 * (3 + 4) - 6$ using *postfix* Notation.

The arithmetic expression $2 * (3 + 4) - 6$ in *postfix* notation is: $2\ 3\ 4\ +\ *\ 6\ -$. Suppose the stack is empty. Read the *postfix* expression from left to right until the end of the expression. It takes the following steps to evaluate:

Input	Stack	Operation	Comment
2	2		Push onto the Stack.
3	2, 3		Push onto the Stack.
4	2, 3, 4		Push onto the Stack.
+	2, 7	$3+4 = 7$	Pop values 4 and 3 from Stack. Perform the addition operation $3 + 4$ and push result 7 onto the Stack.
*	14	$2*7 = 14$	Pop values 7 and 2 from Stack. Perform multiplication

			operation $2 * 7$ and push result 14 onto the Stack.
6	14, 6		Push onto the Stack.
	8	$14 - 6 = 8$	Pop values 6 and 14 from Stack. Perform subtraction operation $14 - 6$ and push result 8 onto the Stack. This is the final result.

Example 4:

Evaluate the *postfix* expression 15, 6, 2, +, *, 4, / using Stack.

Suppose the stack is empty. Read the *postfix* expression from left to right until the end of the expression. It takes the following steps to evaluate:

Input	Stack	Operation	Comment
15	15		Push onto the Stack.
6	15, 6		Push onto the Stack.
2	15, 6, 2		Push onto the Stack.
+	15, 8	$6 + 2 = 8$	Pop values 2 and 6 from Stack. Perform the addition operation $6 + 2$ and push result 8 onto the Stack.
*	120	$15 * 8 = 120$	Pop values 8 and 15 from Stack. Perform multiplication operation $15 * 8$ and push result 120 onto the Stack.
4	120, 4		Push onto the Stack.
/	30	$120 / 4 = 30$	Pop values 4 and 120 from Stack. Perform the division operation $120 / 4$ and push result 30 onto the Stack. This is the final result.

Example 5:

Evaluate the *postfix* expression 6 5 2 3 + 8 * + 3 + * using stack.

Suppose the stack is empty. Read the *postfix* expression from left to right until the end of the expression. It takes the following steps to evaluate:

Input	Stack	Operation	Comment
6	6		Push onto the Stack.
5	6, 5		Push onto the Stack.
2	6, 5, 2		Push onto the Stack.
3	6, 5, 2, 3		Push onto the Stack.
+	6, 5, 5	$2 + 3 = 5$	Pop values 3 and 2 from Stack. Perform the addition operation "2+3" and push result 5 onto the Stack.

8	6, 5, 5, 8		Push onto the Stack.
*	6, 5, 40	$5 * 8 = 40$	Pop 8 and 5 from Stack. Perform multiplication operation "5*8" and push result 40 onto the Stack.
+	6, 45	$5 + 40 = 45$	Pop 40 and 5 from Stack. Perform the addition operation "5+40" and push result 45 onto the Stack.
3	6, 45, 3		Push onto the Stack.
+	6, 48	$45 + 3 = 48$	Pop 3 and 45 from Stack. Perform the addition operation "45+3" and push result 48 onto the Stack.
*	288	$6 * 48 = 288$	Pop 48 and 6 from the Stack. Perform the multiplication operation "6*48" and push result 288 onto the Stack.

Program Code 5.10

// Program to evaluate a *postfix* expression

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
#include <ctype.h>
```

```
#include <math.h>
```

```
// prototype of function to convert a character digit to a numeric digit
int character_to_value(char);
```

```
// prototype of function to perform arithmetic operation
int operation(int, char, int);
```

```
class postfix
```

```
{
```

```
private:
```

```
int stack[50];
```

```
int top;
```

```
public:
```

```
postfix() { top = -1; }
```

```
void scan(char []);
```

```
void push(int);
```

```
int pop(void);
```

```
};
```

```
void main(void)
```

```
{
```

```
postfix s1;
```

```

        clrscr();
        char exp[100];
        cout<<"Enter postfix expression without spaces : ";
        cin>>exp;
        s1.scan(exp);
        getch();
    } //end of main()

    // member function to read an expression
    void postfix::scan(char str_exp[])
    {
        int temp, v1, v2, temp_val, res = 0;
        char ch;
        for (int i = 0; str_exp[i]!='\0'; i++)
        {
            ch = str_exp[i];

            // if ch is digit (operand), convert it into numeric digit and then
            // push onto the stack1 (stack for operands)
            if(isdigit(ch))
            {
                temp_val = character_to_value(ch);
                push(temp_val);
            }

            // if ch is an alphabet (operand), input value for it and then
            // push onto the stack1 (stack for operands)
            else if(isalpha(ch))
            {
                cout<<"Enter value for "<<ch<<" : ";
                cin>>temp_val;
                push(temp_val);
            }

            else if(ch=='^' || ch=='*' || ch=='/' || ch=='+' || ch=='-')
            {
                v1 = pop(); // remove first value from stack
                v2 = pop(); // remove second value from stack
                res = operation(v2, ch, v1);
                push(res);
            }
        }

        cout<<"Final result is "<<pop();
    } //end of scan()

    // member function to push operand onto stack
    void postfix::push(int n)
    {

```



```
    top++;  
    stack[top] = n;  
} //end of push()
```

```
    //member function to remove operands from stack  
int postfix::pop(void)
```

```
{  
    int num;  
    num = stack[top];  
    top--;  
} // end of pop()
```

```
    //definition of function to perform arithmetic operation  
int operation(int x, char op, int y)
```

```
{  
    int res;  
    switch(op)  
    {  
        case '*':    res = x * y; break;  
        case '/':    res = x / y; break;  
        case '+':    res = x + y; break;  
        case '-':    res = x - y; break;  
        case '^':    res = pow(x, y);  
    }  
    return res;  
} // end of operation()
```

```
    //definition of function to convert character digit into numeric digit  
int character_to_value(char x)
```

```
{  
    int val;  
    switch(x)  
    {  
        case '0':    val = 0; break;  
        case '1':    val = 1; break;  
        case '2':    val = 2; break;  
        case '3':    val = 3; break;  
        case '4':    val = 4; break;  
        case '5':    val = 5; break;  
        case '6':    val = 6; break;  
        case '7':    val = 7; break;  
        case '8':    val = 8; break;  
        case '9':    val = 9; break;  
    }  
    return val;  
} // end of character_to_value()
```

Output of the program:

Enter postfix expression without space : AB/C-DE*+
 Enter integer value for A: 12
 Enter integer value for B: 6
 Enter integer value for C: 3
 Enter integer value for D: 4
 Enter integer value for E: 5
 Final result is =19

5.6.4 Evaluation of Prefix Expression

We know that in *postfix* expression operator comes after the operands and in *prefix* expression operator comes before the operands. The algorithm to evaluate a *prefix* expression is similar to evaluate *postfix* expression. There is only one difference between the evaluation of *postfix* expression and *prefix* expression:

- Postfix expression is evaluated from left to right.
- Prefix expression is evaluated from right to left.

Therefore, reverse the *prefix* expression before evaluation and read the reverse *prefix* expression from left to right until the end of the expression.

⚠ The *prefix* expression is evaluated from right to left (instead of left to right), therefore, operation on popped values *V1* and *V2* is performed as follows:

V1 operator V2

However, *postfix* expression is evaluated from left to right, therefore, operation on popped values *V1* and *V2* is performed as follows:

V2 operator V1

Algorithm – Evaluation of Prefix Expression

Write an algorithm to evaluate a *prefix* expression.

- 1- START
- 2- Initialize an empty Stack
- 3- INPUT *prefix* expression
- 4- Reverse *prefix* expression and store it into EXP
- 5- C = 0
- 6- REPEAT Step-7 to 10 WHILE EXP[C] != NULL
 [Read the expression from left to right until the end of the expression]
- 7- CH = EXP[C]
- 8- IF CH is an Operand, then push operand onto the Stack
- 9- IF CH is an Operator, then:
 - i) POP top two operands of the Stack.

- ii) Perform the arithmetic operation on the operands (first top operand, operator, second top operand).
 - iii) PUSH the computed result onto the Stack.
- 10- $C = C + 1$
[End of Step-6 Loop]
 - 11- POP computed result from the Stack.
 - 12- PRINT the computed result.
 - 13- END

Example 1:

Evaluate the *prefix* expression $- * + 4 3 2 5$ using Stack.

Reverse the *prefix* expression $- * + 4 3 2 5$ such as $5 2 3 4 + * -$. Read the reverse *prefix* expression from left to right until the end of the expression. It takes the following steps to evaluate:

Input	Stack	Operation	Comment
5	5		Push onto the Stack.
2	5, 2		Push onto the Stack.
3	5, 2, 3		Push onto the Stack.
4	5, 2, 3, 4		Push onto the Stack.
+	5, 2, 7	$4+3 = 7$	Pop 4 and 3 from the Stack. Perform the addition operation "4+3" and push result 7 onto the Stack.
*	5, 14	$7*2 = 14$	Pop 7 and 2 from the Stack. Perform the multiplication operation "7*2" and push result 14 onto the Stack.
-	9	$14-5 = 9$	Pop 14 and 5 from the Stack. Perform the subtraction operation "14-5" and push result 9 onto the Stack. It is the final result of the given <i>prefix</i> expression.

Example 2:

Evaluate the *prefix* expression $+ 3 + 4 / 20 4$ using Stack.

Scan the *prefix* expression $+ 3 + 4 / 20 4$ from right to left (or reverse the *prefix* expression such as $4 20 / 4 + 3 +$ and read from left to right) until end of the expression. It takes the following steps to evaluate:

Input	Stack	Operation	Comment
4	4		Push onto the Stack.
20	4, 20		Push onto the Stack.
/	5	$20/4 = 5$	Pop 20 and 4 from the Stack. Perform the division operation "20/4" and push result 5 onto the Stack.

4	5, 4		Push onto the Stack.
+	9	$4+5 = 9$	Pop 4 and 5 from the Stack. Perform the addition operation "4+5" and push result 9 onto the Stack.
3	9, 3		Push onto the Stack.
+	12	$3+9 = 12$	Pop 3 and 9 from the Stack. Perform the addition operation "3+9" and push result 12 onto the Stack. It is the final result of the given <i>prefix</i> expression.

Example 3:

Evaluate the *prefix* expression `"/ - * 2 5 * 1 2 - 11 9"` using Stack.

Scan the *prefix* expression `"/ - * 2 5 * 1 2 - 11 9"` from right to left (or reverse the *prefix* expression such as `"9 11 - 2 1 * 5 2 * - /"` and read from left to right) until end of the expression. It takes following steps to evaluate:

Input	Stack	Operation	Comment
9	9		Push onto the Stack.
11	9, 11		Push onto the Stack.
-	2	$11-9 = 2$	Pop 11 and 9 from the Stack. Perform subtraction operation "11-9" and push result 2 onto the Stack.
2	2, 2		Push onto the Stack.
1	2, 2, 1		Push onto the Stack.
*	2, 2	$1*2 = 2$	Pop 1 and 2 from the Stack. Perform multiplication operation "1*2" and push result 2 onto the Stack.
5	2, 2, 5		Push onto the Stack.
2	2, 2, 5, 2		Push onto the Stack.
*	2, 2, 10	$2*5 = 10$	Pop 2 and 5 from the Stack. Perform multiplication operation "2*5" and push result 10 onto the Stack.
-	2, 8	$10-2 = 8$	Pop 10 and 2 from the Stack. Perform subtraction operation "10-2" and push result 8 onto the Stack.
/	4	$8/2 = 4$	Pop 8 and 2 from the Stack. Perform division operation "8/2" and push result 4 onto the Stack. It is the final result of the given <i>prefix</i> expression.

Program Code 5.11

```
// Program to evaluate a prefix expression
#include <iostream.h>
#include <conio.h>
```



```
#include <ctype.h>
#include <string.h>
#include <math.h>

// function prototype for reversing the prefix expression
char * reverse(char *);

// prototype of function to convert a character digit to a numeric digit
int character_to_value(char);

// prototype of function to perform arithmetic operation
int operation(int, char, int);
class prefix
{
    private:
        int stack[50];
        int top;
    public:
        prefix() { top = -1;}
        void scan(char []);
        void push(int);
        int pop(void);
};

void main(void)
{
    prefix s1;
    clrscr();
    char exp[100], rev_exp[100];
    cout<<"Enter prefix expression without spaces : ";
    cin>>exp;
    // convert prefix expression in reverse order and store it into rev_exp
    strcpy(rev_exp, reverse(exp));
    s1.scan(rev_exp);
    getch();
} //end of main()

// member function to read an expression
void prefix::scan(char str_exp[])
{
    int temp, v1, v2, temp_val, res = 0;
    char ch;
    for (int i = 0; str_exp[i]!='\0'; i++)
    {
        ch = str_exp[i];

        // if ch is digit (operand), convert it to numeric digit and then
        // push onto the stack (stack for operands)
```

```

        if(isdigit(ch))
        {
            temp_val = character_to_value(ch);
            push(temp_val);
        }

        // if ch is an alphabet (operand), input value for it and then
        // push onto the stack (stack for operands)
        else if(isalpha(ch))
        {
            cout<< "Enter value for "<<ch<< " : ";
            cin>>temp_val;
            push(temp_val);
        }

        else if(ch=='^' || ch=='*' || ch=='/' || ch=='+' || ch=='-')
        {
            v1 = pop();        // remove first value from stack
            v2 = pop();        // remove second value from stack
            res = operation(v1, ch, v2);
            push(res);
        }

    } // end of for loop that reads the expression
    cout<<"Final result is "<<pop();
} //end of scan()

// member function to push operand onto stack
void prefix::push(int n)
{
    top++;
    stack[top] = n;
} //end of push()

//member function to remove operands from stack
int prefix::pop(void)
{
    int num;
    num = stack[top];
    top--;
    return num;
} //end of pop()

// definition of reverse function to reverse the order of prefix expression
char * reverse(char * p_exp)
{
    char temp_stk[100], temp[100], i, j, t = -1;

```



```
        // push characters of string onto the stack 'temp_stk'
for(i = 0; p_exp[i]!='\0'; i++)
{
    t++;
    temp_stk[t] = p_exp[i];
}

// pop characters of string from stack 'temp_stk' and store in 'temp' in reverse order
for(i = t, j = 0 ; i >= 0; i--, j++)
    temp[j] = temp_stk[i];
temp[j] = '\0';           //append null sign at the end of temp
return temp;
} //end of reverse()
```

```
//definition of function to perform arithmetic operation
int operation(int x, char op, int y)
{
    int res;
    switch(op)
    {
        case '*':    res = x * y; break;
        case '/':    res = x / y; break;
        case '+':    res = x + y; break;
        case '-':    res = x - y; break;
        case '^':    res = pow(x, y);
    }
    return res;
} //end of operation()
```

```
//definition of function to convert character digit into numeric value
int character_to_value(char x)
{
    int val;
    switch(x)
    {
        case '0':    val = 0; break;
        case '1':    val = 1; break;
        case '2':    val = 2; break;
        case '3':    val = 3; break;
        case '4':    val = 4; break;
        case '5':    val = 5; break;
        case '6':    val = 6; break;
        case '7':    val = 7; break;
        case '8':    val = 8; break;
        case '9':    val = 9; break;
    }
}
```

```
return val;
} // end of character_to_value()
```

Output of the program:

Enter prefix expression without spaces : /*25*12-A9
 Enter integer value for A: 11
 Final result is =4

Summary --- Complexities of Stack

Operation	Time Complexity		Space Complexity
	Average-Case	Worst-Case	Worst-Case
Pushing (Insertion)	$\Theta(1)$	$O(1)$	$O(n)$
Popping (Deletion)	$\Theta(1)$	$O(1)$	$O(n)$

Short Answers to the Questions**What is a stack?**

* A stack is a linear data structure in which items are accessible only from one end. This end is called the *top* of the stack. When a new item is added in a stack, it is placed on top of all the previously added items. The items are removed in the reverse order in which they are added i.e. top of the item is removed first. Therefore the items in a stack are stored and retrieved in "Last In First Out - (LIFO)" manner. It means that items added at last are retrieved first. For this reason, a stack is also known as *Last-In, First-Out (LIFO)* data structure.

What is the difference between push and pop operations?

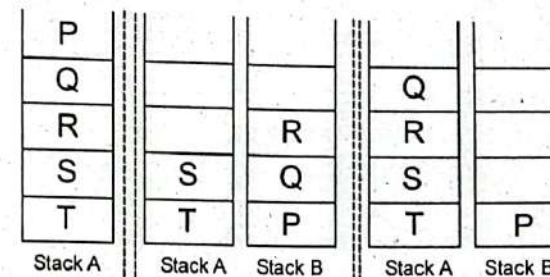
When a data item is added or inserted into a stack at its top position, the operation is called *push operation*. When a data item is removed from the stack, the operation is called *pop operation*.

Consider the following scenario:

1. The five items named P, Q, R, S, and T have inserted into stack A one after the other, starting from T in reverse order.
2. The stack is popped three times and each element is inserted into another stack B.
3. Then two elements are deleted from the stack B and pushed back onto the stack A.

At this stage, what are the topmost elements of stack A and stack B respectively?

The topmost element of stack A will be Q, and the topmost element of stack B will be P, as shown here.

**EXERCISE****Q#1 Select the correct answer from the given multiple choices.**

1. The end of a stack from where data items can be inserted and deleted is called:
 - (a) top of the stack
 - (b) bottom of the stack
 - (c) head of the stack
 - (d) None
2. How many basic operations can be performed on a stack?
 - (a) 4
 - (b) 5
 - (c) 2
 - (d) Many
3. Stack is also called:
 - (a) First-In Last-Out (FILO)
 - (b) Last-In First-Out (LIFO)
 - (c) First-In First-Out (FIFO)
 - (d) Last-In Last-Out (LILO)
4. Which one operation is used to insert new item into stack?
 - (a) Add
 - (b) Insert
 - (c) Pop
 - (d) Push
5. Which one operation is used to pick or remove an item from stack?
 - (a) Add
 - (b) Insert
 - (c) Pop
 - (d) Push
6. Which one indicates the polish notation?
 - (a) $(X+Y)*Z$
 - (b) $*+XYZ$
 - (c) $X+Y*Z$
 - (d) None
7. Stack data structure is:
 - (a) FIFO
 - (b) LILO
 - (c) LIFO
 - (d) None
8. In the stack, the process of inserting an element in the stack is called as:
 - (a) Pop
 - (b) Evaluation
 - (c) Create
 - (d) Push

9. The process of removing an element from the stack is called as:
 - (a) Postfix Expression
 - (b) Pop
 - (c) Create
 - (d) Push
10. In the stack, if user tries to remove element from the empty stack then the error displayed is:
 - (a) Empty Collection
 - (b) Underflow of Stack
 - (c) Garbage Collection
 - (d) Overflow of Stack
11. User pushes 1 element in the stack having already five elements and having stack size as 5 then the stack becomes:
 - (a) Overflow
 - (b) Underflow
 - (c) User Flow
 - (d) Crash
12. What will be the postfix expression for the following infix expression: $B * C + D / E$?
 - (a) $B * C D E / +$
 - (b) $B C D * E / +$
 - (c) $B C * D E / +$
 - (d) $B C * D E + /$
13. What will be the postfix expression for the following infix expression: $A / B ^ C - D$?
 - (a) $A B / ^ C D -$
 - (b) $A B ^ / C D -$
 - (c) $A B C ^ / D -$
 - (d) $A B C / ^ D -$
14. What will be the postfix expression for the following infix expression: $A + B * C ^ D$?
 - (a) $A B C D * + ^$
 - (b) $A B C + D * ^$
 - (c) $A B C D + * ^$
 - (d) $A B C D ^ * +$
15. Evaluate postfix expression from the following infix expression: $A + B * (C + D) / F + D * E$.
 - (a) $A B + C D * F / + D * E$
 - (b) $A B C D + * F / + D E * +$
 - (c) $A B C D + * / F + D E *$
 - (d) $A B + C D * F / + D E *$
16. Expression in which operator is written after operand is called as:
 - (a) Infix Expression
 - (b) Postfix Expression
 - (c) Prefix Expression
 - (d) Pastfix Expression
17. What will be the postfix form of the above expression: $(A + B) * (C * D - E) * F / G$?
 - (a) $A B + C D * E - F G / * *$
 - (b) $A B + C D * E - F G * / *$
 - (c) $A B + C D E * - F G / * *$
 - (d) $A B + * C * D E - * F G /$
18. If the sequence of operations on a stack is:
 - push(1)
 - push(2)
 - pop
 - push(1)
 - push(2)
 - pop
 - pop
 - pop
 - push(2)
 - pop

Then the sequence of popped out values are:

- (a) 2, 2, 1, 2, 2 (b) 2, 2, 1, 1, 2 (c) 2, 1, 2, 2, 1 (d) 2, 1, 2, 2, 2
19. Stack cannot be used for:
- (a) Evaluation of Expression in Postfix form
 (b) Reversing String
 (c) Allocating Resources and Scheduling
 (d) Implementation of Recursion
20. A postfix expression is just the reverse of the prefix expression:
- (a) True (b) False

Answers of MCQs

01 (a)	02 (c)	03 (b)	04 (d)	05 (c)	06 (b)	07 (c)	08 (d)	09 (b)	10 (b)
11 (a)	12 (c)	13 (c)	14 (d)	15 (b)	16 (b)	17 (d)	18 (b)	19 (c)	20 (a)

- Q.2 What is a stack? How is it implemented in computer memory?
- Q.3 What are push & pop operations on the stack? Explain with examples.
- Q.4 Show the sequence of operations to solve the following given expression using Stack:
- $$7 + (8 * 4) / 2$$
- Q.5 Convert the following *infix* expressions into postfix and prefix notations:
- i) $A + 2 * A / B + B^2$ ii) $A + 3 * (B + C)^2$
 iii) $A + B * C + (D * E + F) * G$ iv) $(A + B) * C$
 v) $A * B + C / D$ vi) $A * (B + C / D)$
 vii) $L + K / (A + B) * M - (F + G)$ viii) $(A + B^C) * D + E^5$
- Q.6 Write a program that inputs an integer value and converts it into octal number and hexadecimal number using stack.
- Q.7 Write a program that inputs 15 values and pushes them into a stack, then pops the values from the stack and displays only odd values on the screen.
- Q.8 Evaluate the following *postfix* expressions using Stack:
- i) $4 \ 5 \ + \ 7 \ 2 \ - \ *$
 ii) $4 \ 3 \ 2 \ 5 \ * \ - \ +$
 iii) $5 \ 3 \ 2 \ * \ 8 \ + \ *$
 iv) $2 \ 3 \ 4 \ + \ * \ 6 \ -$
 v) $1 \ 2 \ 3 \ * \ + \ 4 \ -$

vi) $5\ 1\ 2\ +\ 4\ *\ +\ 3\ -$

vi) $10,\ 5,\ 3,\ +,\ *,\ 2,\ /$

Source Code of Programs

Source code of programs given in "DATA STRUCTURES USING C++" can be received by sending an e-mail to "pakman_series@hotmail.com".

RECURSION

Recursion is a powerful programming technique for solving problems. It involves breaking down a problem into smaller sub-problems until a small enough problem is obtained that can be solved trivially. Usually, recursion involves a function calls itself again and again either directly or indirectly till the final calculation of the result is obtained. The involved function in recursion is called a *recursive function*. The number of times a function is called recursively is called *depth of recursion*.

Recursion is a fundamental concept in computer science. It was not supported in older programming languages like FORTRAN, COBOL, and BASIC. However, all the latest modern programming languages (such as C++ and Java) support recursion.

Recursive Function

A function that calls itself during its execution is called a *recursive function*. This enables function to repeat itself several times until the final result is obtained. An intermediate result obtained in each iteration. Recursive functions are commonly used in programming because they reduce the size of programs.

Conditions for Recursive Function

There are two important conditions that must be satisfied by any recursive function. These conditions are necessary to prevent the recursive function from running infinitely. These conditions are as follows:

- i) **Base Criteria:** A recursive function must have a non-recursive part, called the *base criteria*. It is a condition used to terminate the execution of the recursive function. When this condition is reached, the recursive function stops calling itself recursively.
- ii) **Progressive Approach:** The recursive calls should progress in such a way that each time the recursive function is called, the condition must get closer to the *base criteria*. Therefore, the function must be called with different parameters.

A recursive function that fulfills these two conditions is called a *well-defined recursive function*.

Consider the following recursive function that finds the factorial of a number 'n':

```
int fact(int n)
{
```



```

    if (n <= 1)                // Base criteria
        return 1;
    else
        return n*fact(n-1);    // Recursive Part
}

```

In the above code, base criteria ' $n \leq 1$ ' is defined and the factorial of the number ' n ' can be computed by converting it into a smaller one, till base criteria is reached.

6.1.2 Implementation of Recursive Function through Stacks

A recursive function can be executed several times depending on the situation. It receives values from the calling function and transmits results back to it. For its proper functioning, it must save the parameters and variables upon execution and restore these parameters and variables at completion. While returning values, the function must keep track of the return address in the calling function. This return address is used to transfer the control back to its proper place in the calling function. It saves the return address of each calling function in the same order in which it is called and returns the control to the proper place in the calling function each time the control is returned.

The recursive function terminates execution when the base criteria is reached. Then it returns parameters and variables to the calling function. It returns these values such that the values from the last execution are returned first. This last-in and first-out (LIFO) characteristic of a recursive function shows that stack is the most suitable data structure for its implementation. At each function call, the stack is pushed to save the necessary values. Upon exit from the function, the stack is popped to retrieve the values.

The general algorithm for a recursive function contains the following three parts:

Prologue	It is the opening part of the recursive function. Its purpose is to save the formal parameters, local variables, and return address.
Body	It contains a definition of the function, including the test criteria, in which the function calls itself. Each time the function calls itself, the prologue of the function saves all necessary information required for its functioning.
Epilogue	It is the last part of the recursive function. It restores the most recently saved parameters, local variables, and return addresses.

Example:

Following steps describe the procedure to calculate the factorial of number 3 using the recursive definition:

IF $n = 0$ THEN $n! = 1$

IF $n > 0$ THEN $n! = n \cdot (n-1)!$

Step 1 $3! = 3 \cdot 2!$

Step 2 $2! = 2 \cdot 1!$

Step 3 $1! = 1 \cdot 0!$

Step 4 $0! = 1$

Step 5 $1! = 1 \cdot 1 = 1$

Step 6 $2! = 2 \cdot 1 = 2$

Step 7 $3! = 3 \cdot 2 = 6$

- ★ From steps 1 to 3, the factorial is evaluated in terms of the factorial of another integer.
- ★ Step-4 explicitly evaluates factorial of 0 as 1. Therefore '0!' is the base value of the recursive definition.
- ★ After reaching the base criteria, the evaluation is performed in the reverse order.
- ★ The factorial of 0 is used to find out the factorial of 1. Similarly, 1! is used to find out 2! and so on.

6.1.3 Recursion in Math

Mathematical formulae are often expressed recursively. For example, the definition of N! (factorial of N) is defined as the product of all integers between 1 and N. Therefore, the factorial of 4 is mentioned as:

$$4! = 4 \times 3 \times 2 \times 1 = 24$$

The definition of N! can be expressed recursively as:

$$1! = 1$$

$$N! = N \cdot (N - 1)!$$

The base case of this definition is 1! which is defined to be 1. All other values of N! are defined recursively as N times the value (N - 1)!. Therefore, 5! is equal to 5*4! and 4! is equal to 4*3! and so on. This process continues until we get the base case of 1.

6.1.4 Recursive Programming

The concept of recursive programming is different from that of other programming approaches. A simple mathematical operation is given below to explain the concept of recursive programming.

For example, the sum of values between 1 and 10 is equal to 10, plus the sum of values between 1 and 9. Continuing this idea, the sum of values between 1 and 9 is equal to 9, plus the sum of values between 1 and 8 and so on.

In C++ (and many other programming languages), a function can call itself. Each call to the function creates a new computational environment. For example, to compute the sum of numbers between 1 and N, the recursive mathematical technique is given as under:

$$\begin{aligned}
 \sum_{i=1}^n &= N + \sum_{i=1}^{n-1} i \\
 &= N + N - 1 + \sum_{i=1}^{n-2} i \\
 &= N + N - 1 + N - 2 + \sum_{i=1}^{n-3} i \\
 &= N + N - 1 + N - 2 + \dots + 2 + 1
 \end{aligned}$$

When a function calls itself, all local variables and parameters are newly defined and a separate space is allocated on the stack. The function is executed with new parameter values from the beginning. A recursive call does not make a new copy of the function but only the arguments are newly defined. As each recursive call returns, the local variables and parameters of the previous function call are removed from the stack and new variables and parameter values are added into it.

A recursive function to compute the sum of values between 1 and N can be defined as follows:

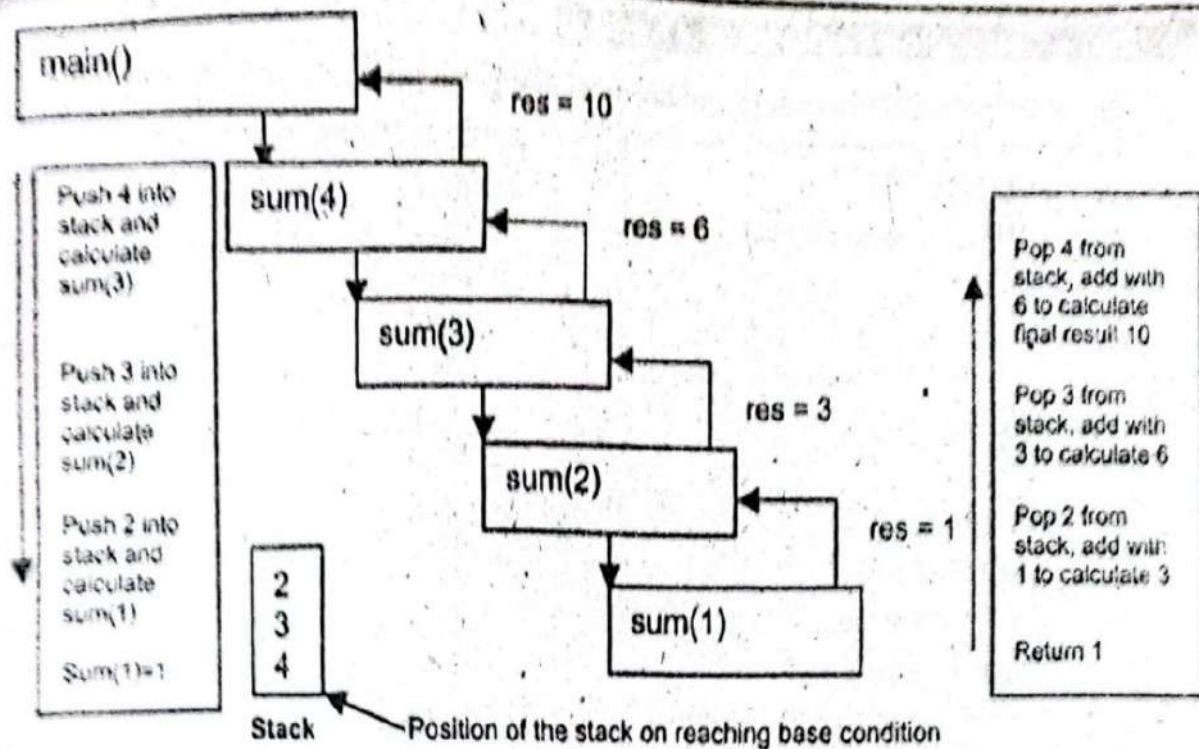
```

int sum(int N)
{
    if (N == 1)
        return 1;
    else
        return = N + sum(N - 1);
}

```

In the above recursive function definition, the sum of the numbers between 1 and N is equal to N, plus the sum of numbers between 1 and N - 1. The parameter passed to the function "sum" is decremented each time the function is called, so that it will get closer to the base case, i.e. 1. At each recursion, the intermediate value is pushed into the stack. When the parameter value becomes equal to 1, the calling process of the recursive function is stopped. The recursion begins to fold back into the earlier versions of the function "sum" by popping stack values to calculate the results and returning the calculated value each time. Each returned value contributes to the computation of the sum at a higher level.

The figure given below shows the recursive process when the main() function calls the function "sum" to compute the sum of the values between 1 and 4. Each box represents the function call with a new variable and parameter (N-1). When the base case is reached then the calls begin to return their results.



The steps to compute the sum of integers between 1 and 4 using the recursive definition are described below:

IF $N = 1$ THEN return 1

ELSE

Step-1: sum between 1 and 4 = $4 + \text{sum between 1 and 3}$

Step-2: sum between 1 and 3 = $3 + \text{sum between 1 and 2}$

Step-3: sum between 1 and 2 = $2 + \text{sum between 1 and 1}$

Step-4: sum between 1 and 1 = return 1

Step-5: return $2 + 1 = 3$

Step-6: return $3 + 3 = 6$

Step-7: return $4 + 6 = 10$

From Step-1 to 3, the values in the stack will be pushed in the following order:

- ★ 4 will be pushed in step-1.
- ★ 3 will be pushed in step-2.
- ★ 2 will be pushed in step-3.

At step-4, the base case has finally reached and value 1 is returned. Therefore, the evaluation is performed in the reverse order as follows:

- The value 2 will be popped from the stack and added to 1. The result will be 3.
- The value 3 will be popped from the stack and it is added to 3. The result will be 6.
- The value 4 will be popped from the stack and it is added to 6. The result will be 10. It will be the final result.

Algorithm – Computing Factorial of a Number

Write an algorithm that finds the factorial of a given integer using a loop statement

- 1- START
- 2- INPUT integer number in variable N
- 3- IF N = 0
 Factorial = 1
 PRINT "Factorial = ", Factorial
 RETURN
 END IF
- 4- Factorial = 1
 REPEAT FOR C = 1 To N BY 1
 Factorial = Factorial * C
 [End of Loop]
- 5- PRINT "Factorial = ", Factorial
- 6- END

Program Code 6.1

// Program to find the factorial of a given number using loop statement

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
class factorial
```

```
{
```

```
public:
```

```
    //member function to compute factorial
```

```
    int Fact(int n)
```

```
    {
```

```
        int f = 1;
```

```
        if(n==0)
```

```
            return f;
```

```
        else
```

```
            for( ; n>1; n--)
```

```
                f = f * n;
```

```
        return f;
```

```
    } //end of Fact()
```

```
};
```

```
void main(void)
```

```
{
```

```
    factorial obj;
```

```
    int val, res;
```

```

clrscr();
cout<<"Enter value to find factorial : " ;
cin>>val ;
res = obj.Fact(val);
cout<<"Factorial of "<<val<<" is = "<<res;
getch();
} //end of main()

```

Output of the program:

Enter value to find factorial : 6
Factorial of 6 is = 720

Function : Factorial (F, N)

Write a function sub-algorithm that finds the factorial of a given number using recursive function.

FACTORIAL(F, N)

```

1-   IF N = 0 THEN
        F = 1
        RETURN
    END IF
2-   CALL FACTORIAL(F, N-1)
    F = F * N
    RETURN

```

In the above sub-algorithm, the function FACTORIAL calls itself to calculate the factorial of the given integer N.

Program Code 6.2

```

// Program to find the factorial of a given number using recursion
#include <iostream.h>
#include <conio.h>

class factorial
{
    public:
        //member function to compute factorial
        long Fact(int n)
        {
            int res;
            if (n==1)

```



```

        return 1;
    else
        res = n * Fact(n-1);
    return res;

} //end of Fact()

};

void main(void)
{
    factorial obj;
    int val, res;
    clrscr();
    cout<<"Enter value to find factorial : ";
    cin>>val;
    res = obj.Fact(val);
    cout<<"Factorial of "<<val<<" is = "<<res;
    getch();
} //end of main()

```

Output of the program:

Enter value to find factorial : 4
Factorial of 4 is = 24

Now suppose we modify the recursion function Fact() used in the program code 6.2, given in the following program. Then observe the output of the program.

Program Code 6.3

```

// Program to find the factorial of a given number using recursion

#include <iostream.h>
#include <conio.h>
class factorial
{
public:
    //member function to compute factorial
    long Fact(int n)
    {
        int res;
        cout<<endl<<"Fact() has been called with n = " <<n;
        if (n==1)
            return 1;
        else
            res = n * Fact(n-1);
        cout<<endl<<"Intermediate result for " <<n
            <<" * Fact(" <<n-1<<") : " <<res;
    }
};

```

```

        return res;
    } //end of Fact()

};

void main(void)
{
    factorial obj;
    int val, res;
    clrscr();
    cout<<"Enter value to find factorial : ";
    cin>>val;
    res = obj.Fact(val);
    cout<<endl<<"Factorial of "<<val<<" is = "<<res;
    getch();
} //end of main()

```

Output of the program:

Enter value to find factorial : 4

Fact() has been called with n = 4

Fact() has been called with n = 3

Fact() has been called with n = 2

Fact() has been called with n = 1

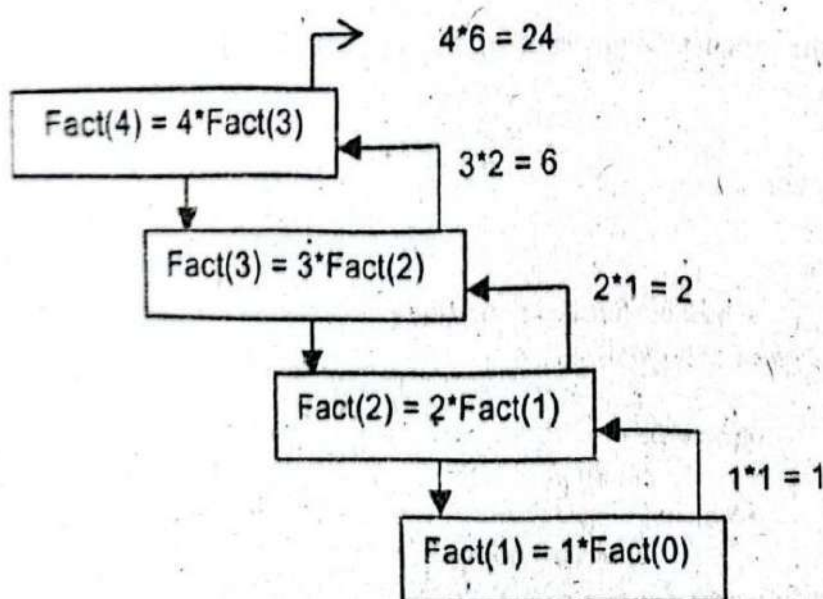
Intermediate result for $2 * \text{Fact}(1)$: 2

Intermediate result for $3 * \text{Fact}(2)$: 6

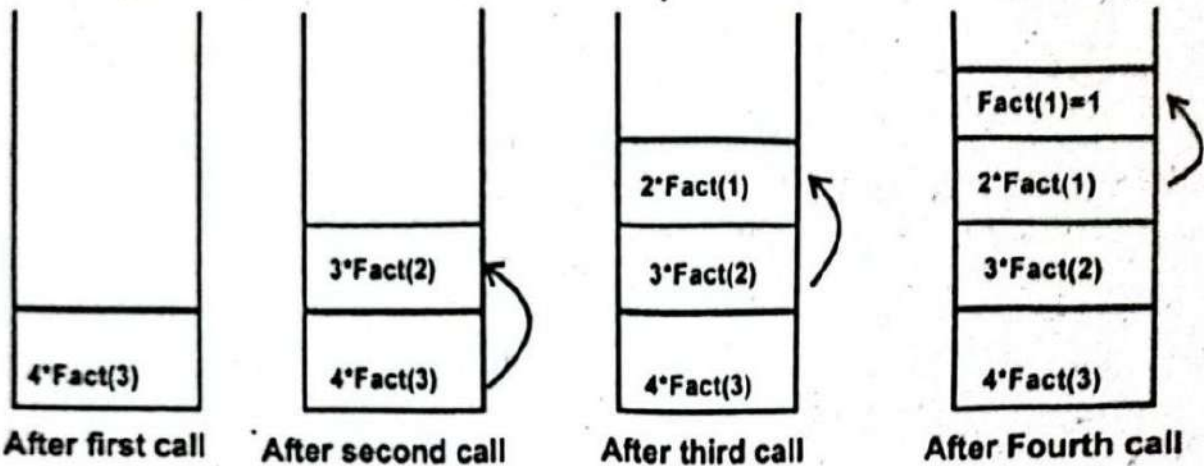
Intermediate result for $4 * \text{Fact}(3)$: 24

Factorial of 4 is = 24

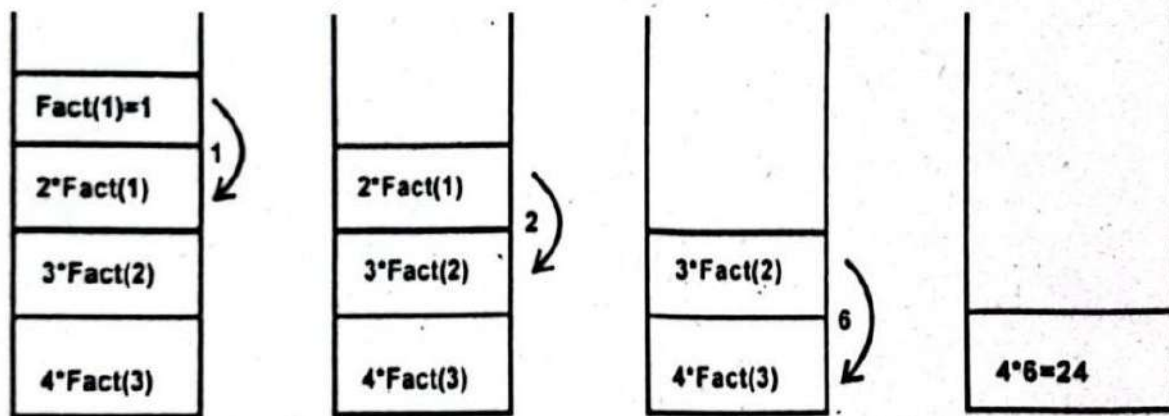
Suppose the factorial of 4 is to be calculated. The recursive process to find its factorial is run below:



The following figures show the stack position when function Fact() is called for values 3, 2, and 1 respectively:



The following figures show the stack position by returning values from base case to calling function:



Program Code 6.4

```
// Program to find the exponential power of an integer using recursion
#include <iostream.h>
#include <conio.h>

class exponent
{
public:
    // member function to find the exponential power
    int power(int b, int n)
    {
        if(n == 0)
            return 1;
        else
            return b * power(b, n-1);
    } //end of power()
};
```

```

void main(void )
{
    exponent obj;
    int val, p, res;
    clrscr();
    cout<<"Enter an integer value : ";
    cin>>val;
    cout<<"Enter value for exponent : ";
    cin>>p;
    res = obj.power(val, p);
    cout<<"Result is = "<<res;
    getch();
} //end of main()

```

Output of the program:

```

Enter an integer value : 2
Enter value for exponent : 3
Result is = 8

```

Function : gcd (m, n)

Write a function sub-algorithm that finds the greater common divisor of given numbers using recursive function.

GCD(M, N)

```

1-   IF M < N THEN
        CALL GCD(N, M)
        RETURN
    END IF
2-   R = M % N
    IF R == 0 THEN
        RETURN N
    ELSE
        CALL GCD(N, R)
        RETURN
    END IF

```

In the above sub-algorithm, the function GCD calls itself to calculate the gcd of the given integers M, N. It has a great application in security algorithms to set encryption and decryption keys.

Program Code 6.5

```

// Program to find the gcd of a given number using recursion

#include <iostream.h>
#include <conio.h>

```



```

class GCD
{
    public:
        // member function to find the gcd
        int find_gcd(int m, int n)
        {
            int r;
            if (m < n)
                return find_gcd(n, m);
            r = m % n;
            if (r == 0)
                return n;
            else
                return find_gcd(n, r);
        } //end of find_gcd()
};

void main(void)
{
    GCD obj;
    int m, n;
    clrscr();
    cout<<"Enter an integer value : ";
    cin>>m;
    cout<<"Enter second integer value : ";
    cin>>n;
    cout<<"GCD of "<<m<<"and "<<n<<" = "<<obj.find_gcd(m,n);
    getch();
} //end of main()

```

Output of the program:

```

Enter an integer value : 15
Enter second integer value : 25
GCD of 15 and 25 = 5

```

Function : Tower of Hanoi

Write a function sub-algorithm that solves the game Tower of Hanoi using recursive function.

MoveTower(disk, source, dest, spare)

IF disk == 0 THEN

Move disk from source to dest

ELSE

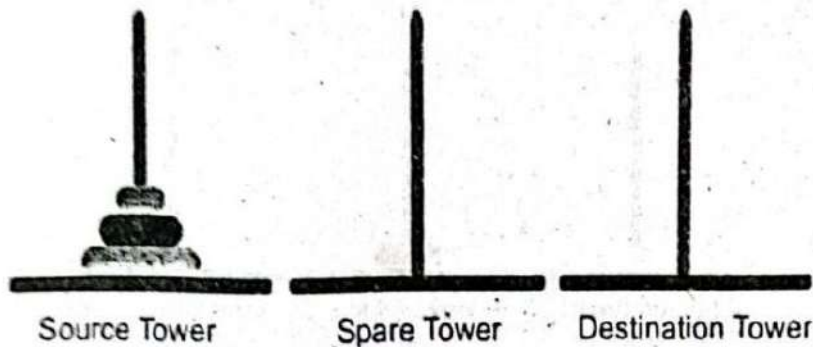
CALL MoveTower (disk-1, source, spare, dest)

Move disk from source to dest (i.e. destination)

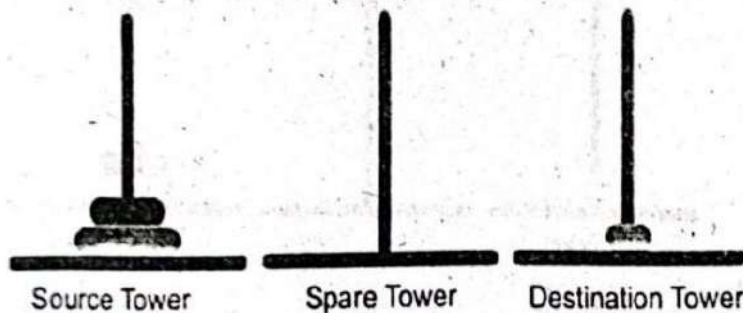
CALL MoveTower (disk-1, spare, dest, source)

END IF

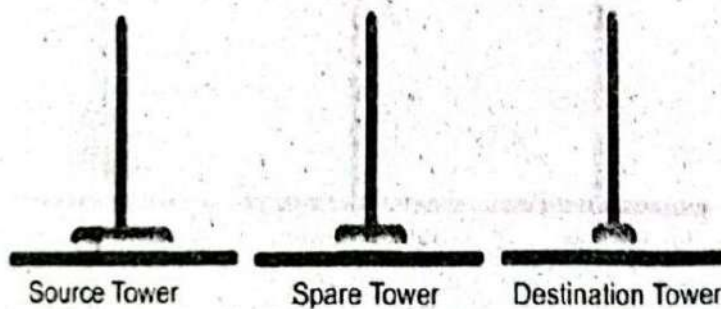
In the above sub-algorithm, the function MoveTower calls itself to solve the Tower of Hanoi game. The objective of the game is to move all the disks over to the destination tower using a spare tower. But larger disk cannot be placed over a smaller one. An example is given below to clear the concept of this game, in which 3 disks are given in the source tower. The target is to place them in the same order to destination tower one by one without violating the rule that a larger disk cannot be placed over a smaller one.



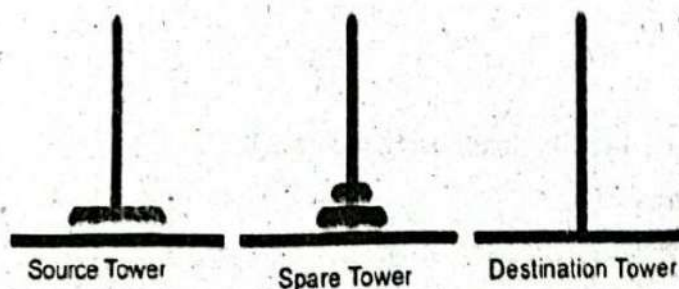
Move -1:

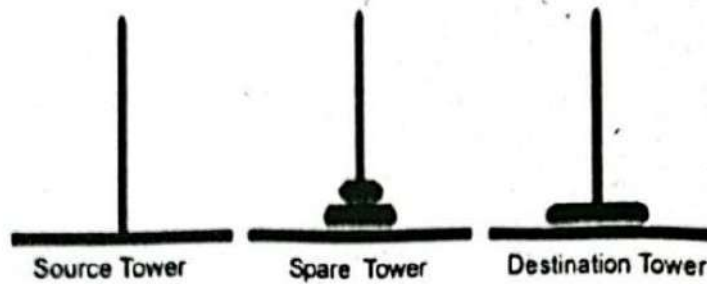
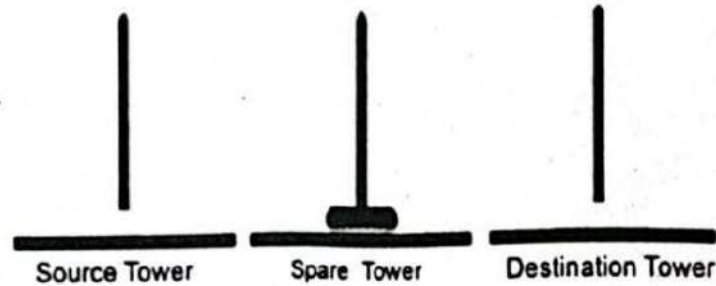
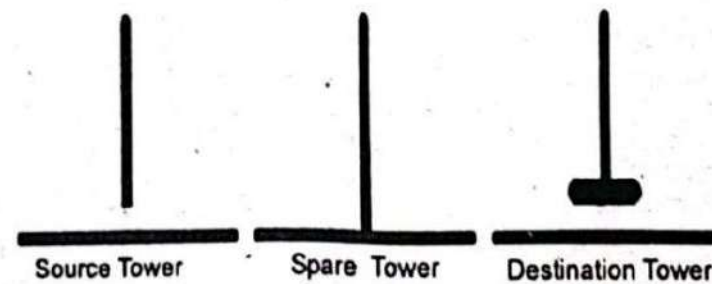
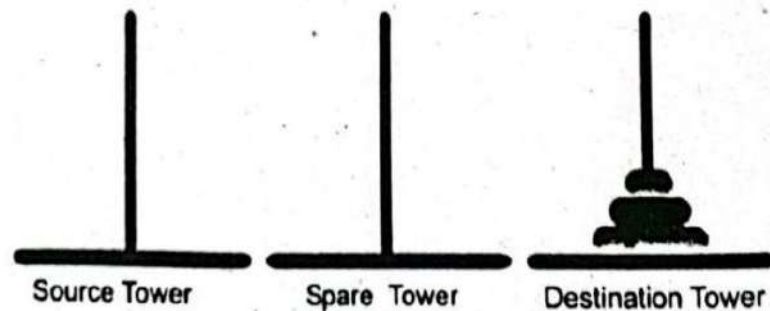


Move -2:



Move -3:



Move -4:**Move -5:****Move -6:****Move -7:**

Target is achieved at the seventh move i.e. Move-7. All the disks are accurately placed on the destination tower.

Program Code 6.6

```
// Program for Tower of Hanoi using recursion
#include<iostream.h>
#include<conio.h>
```

```
//Tower of Hanoi function implementation
void TOH(int n, char Sour, char Aux, char Des)
{
    if(n==1)
    {
        cout<<"Move Disk "<<n<<" from "<<Sour<<" to "<<Des<<endl;
        return;
    }

    TOH(n-1,Sour,Des,Aux);
    cout<<"Move Disk "<<n<<" from "<<Sour<<" to "<<Des<<endl;
    TOH(n-1,Aux,Sour,Des);
}

void main(void)
{
    int n;
    clrscr();
    cout<<"Enter no. of disks:";
    cin>>n;

    //calling the TOH
    TOH(n,'A','B','C');
    getch();
}
```

Output of the program:

```
Enter no. of disks:3
Move Disk 1 from A to C
Move Disk 2 from A to B
Move Disk 1 from C to B
Move Disk 3 from A to C
Move Disk 1 from B to A
Move Disk 2 from B to C
Move Disk 1 from A to C
```

6.2 Types of Recursion

There are various types of recursions. Some important types of recursions are as follow

- i) Direct and Indirect Recursions
- ii) Nested and Non-Nested Recursions
- iii) Tail and Non-Tail Recursions
- iv) Linear and Tree Recursions
- v) Excessive Recursion

6.2.1 Direct and Indirect Recursions

When a function calls itself, then this type of recursion is called *direct recursion*. In this type of recursion, only one function is involved. All the above-given program examples use the direct recursion method such as finding factorial of an integer, exponential power of an integer, GCD, and Tower of Hanoi.

When a function is called not by itself but by another function, then this type of recursion is called *indirect recursion*. For example, if function A calls function B and then function B calls back to function A, this is a case of indirect recursion. Chains of many functions are possible. For example, function A calls function B, function B calls function C, and function C calls function A again. An indirect recursion is also known as mutual recursion in which two functions call each other again and again. For example, function A calls another function B which in turn calls A. The depth of indirect recursion may vary.

Program Code 6.7

// Program to print natural numbers from 1 to 10 using indirect recursion

```
#include <iostream.h>
#include <conio.h>
```

```
class AB
{
```

```
public:
```

```
// This function prints n, increments n by 1 and calls B
```

```
void A(int n)
```

```
{
```

```
if (n <= 10)
```

```
{
```

```
cout<<n<<" ";
```

```
n++;
```

```
B(n);
```

```
}
```

```
else
```

```
return;
```

```
}
```

```
// This function prints n, increments n by 1, and calls A
```

```
void B(int n)
```

```
{
```

```
if (n <= 10)
```

```
{
```

```
cout<<n<<" ";
```

```
n++;
```

```
A(n);
```

```
}
```

```
else
```

```
return;
```

```

    };
}

void main(void)
{
    AB obj;
    clrscr();
    obj.A(1);
    getch();
}

```

Output of the program:

```
1 2 3 4 5 6 7 8 9 10
```

Program Code 6.8

// Program to print even numbers from 1 to 10 using indirect recursion

```

#include <iostream.h>
#include <conio.h>

class even_num
{
public:
    // This function stops the recursion process if value of n is greater
    // than or equal to 10 and calls even() with increment by 1
    void check(int n)
    {
        if (n >= 10)
            return;
        else
            even(n+1);
    }

    // This function prints even number and calls back check function
    // with increment by 1
    void even (int n)
    {
        cout<<n<<" ";
        check(n+1);
    }
};

void main(void)
{
    even_num obj;
    clrscr();
    obj.check(1);
    getch();
}

```


}

Output of the program:

2 4 6 8 10

6.2.2 Nested and Non-Nested Recursions

A recursion is said to be a *nested recursion* when a recursive function has a parameter defined in terms of itself. The best example of a nested function is the Ackerman function that grows faster than a multiple exponential function.

```
int Ackermann(int x, int y)
{
    // Base or Termination Condition
    if(x == 0)
        return y + 1;

    // Error Handling condition
    if (x < 0 || y < 0)
        return -1;

    // Recursive call by Linear method
    else if (x > 0 && y == 0)
        return Ackermann(x-1, 1);

    // Recursive call by Nested method
    else
        return Ackermann(x-1, Ackermann(x, y-1));
}
```

In the above code, the Ackermann function is used as a parameter in the fourth return statement which is the form of nested recursion.

A recursion is said to be a *non-nested recursion* when a recursive function has independent parameters. A recursive function for calculating the factorial of a given number is an example of a non-nested function.

6.2.3 Tail and Non-Tail Recursions

Tail recursion is the special case of recursion in which the recursive call is the last thing the function does. It executes one recursive call if and only if there is no pending operation after that call. Often, the value of the recursive call is returned. As such, tail-recursive functions can often be easily implemented iteratively: by taking out the recursive call and replacing it with a loop, the same effect can generally be achieved. It tries to decrease the depth of the stack.

It is desirable to have tail-recursions because the amount of information that gets stored during computation is independent of the number of recursive calls. A good compiler can

optimize tail recursion and convert it to iteration to optimize the performance of the code. Tail recursion is important in languages like Prolog and Functional languages like Clean, Haskell, Erlang, and SML that do not have explicit loop constructs (loops are simulated by recursion).

An example of a tail-recursive function is a function to compute the GCD (Greatest Common Divisor) of two numbers:

```
int gcd(int m, int n)
{
    int r;
    if (m < n)
        return gcd(n,m);
    r = m%n;
    if (r == 0)
        return(n);
    else
        return(gcd(n,r));
}
```

A recursion is said to be *non-tail recursion* if there are one or more pending operations after the recursive call. Consider the following recursion function to calculate the factorial of a given number 'n':

```
long fact(int n)
{
    if (n==0)
        return 1;
    else
        return n*fact(n-1);
}
```

Following code calculates the factorial of a given number 'n' by tail recursion method:

```
long fact (int n)
{
    return tail_fact (n, 1);
}

long tail_fact (int n, int multiplier)
{
    if (n == 0)
        return multiplier;
    else
        return tail_fact (n-1, n * multiplier);
}
```


6.2.4 Linear and Tree Recursions

Linear recursion is the most common type of recursion. In this type of recursion, the recursive function only makes a single call to itself each time the function runs. The factorial function is one of the forms of linear recursion.

A recursion is said to be a tree recursion when the pending operation involves another recursive call. The recursive function calls itself multiple times within an execution. The Fibonacci function is one of the forms of tree recursion.

```
int fibo(int n)
{
    if (n == 0 || n == 1)
        return n;
    else
        return fibo(n - 1) + fibo(n - 2);
}
```

In the above example, the function 'fibo' returns two itself calls. It is said to be a tree recursive method.

6.2.5 Excessive Recursion

A recursion is said to be excessive recursion when a recursive function repeats computations for some parameter values. In excessive recursion, the computations of parameters are done again and again. The same calculation is repeated many times since the system is not remembering what all things already have been computed.

For example, in Fibonacci function as given above, the call fibo(6) results in many repetitions of fibo(4), fibo(3), fibo(2), and fibo(1).

```
fibo(6) will return fibo(5) + fibo(4)
fibo(5) will return fibo(4) + fibo(3)
fibo(4) will return fibo(3) + fibo(2)
fibo(3) will return fibo(2) + fibo(1)
fibo(2) will return fibo(1) + fibo(0)
fibo(1) will return 1      ... and so on
```

It is known to be excessive recursion in which same recursions makes to happen again and again.

6.3 ADVANTAGES AND DISADVANTAGES OF RECURSION

Advantages of Recursion

Following are some important advantages of recursion:

- i) Using recursion we can avoid the unnecessary calling of functions.
- ii) Using recursion, the length of the program can be reduced.
- iii) Complex case analysis and nested loops can be avoided.
- iv) The recursion is very flexible in a data structure like stacks, queues, linked lists, and quicksort.
- v) Recursion is a useful way of defining objects that have a repeated similar structural form.

Disadvantages of Recursion

Following are some disadvantages of recursion:

- ❖ The recursive calls and automatic variables are stored on the stack. For every recursive call, separate memory is allocated to automatic variables with the same name. Therefore, a recursive function requires extra storage space. If recursion is too deep, then there is a danger of running out of space on the stack and ultimately program crashes.
- ❖ The recursion function is not efficient in execution speed and time.
- ❖ The recursive solution is always logical and it is very difficult to trace (debug and to understand).

Short Answers to the Questions

Define the recursive function.

A function that calls itself during its execution is called a *recursive function*. This enables the function to repeat itself several times until the final result is obtained.

What are conditions for a recursive function?

Following are conditions for a recursive function:

- ★ A recursive function must have a non-recursive part, called the *base criteria*. It is a condition used to terminate the execution of the recursive function.
- ★ Each time the recursive function is called directly or indirectly, the condition must get closer to the *base criteria*.

What is the difference between direct and indirect recursions?

When a function calls itself, then this type of recursion is called *direct recursion*. In this type of recursion, only one function is involved. When a function is called not by itself but by another function, then this type of recursion is called *indirect recursion*. In this type of recursion, more than one function are involved.

What is nested recursion?

A recursion is said to be a *nested recursion* when a recursive function has a parameter defined in terms of itself. The best example of a nested function is the Ackerman function that grows faster than a multiple exponential function.

What is linear recursion?

In this type of recursion, the recursive function only makes a single call to itself each time the function runs. The factorial function is one of the forms of linear recursion.

What is tree recursion?

A recursion is said to be a tree recursion when the pending operation involves another recursive call. The recursive function calls itself multiple times within an execution. The Fibonacci function is one of the forms of tree recursion.

EXERCISE

Q#1 Select the correct answer from the multiple choices.

- Recursive problems are implemented by:
(a) Queues (b) Stacks (c) linked lists (d) strings
- The number of times a function is called recursively is called:
(a) Base criteria (b) Recursion life time (c) Depth of recursion (d) None
- A criteria or condition used to terminate the execution of the recursive function is called:
(a) Base criteria (b) Prologue (c) Epilogue (d) None
- Which one part of recursive function saves formal parameters and local variables?
(a) Base criteria (b) Prologue (c) Epilogue (d) None
- Which one is the last part of recursive function?
(a) Body (b) Prologue (c) Epilogue (d) None
- Recursion is a method in which the solution of a problem depends on:
(a) Larger instances of different problems (b) Larger instances of the same problem
(c) Smaller instances of the same problem (d) Smaller instances of different problems
- Which of the following problems can be solved using recursion?
(a) Factorial of a number (b) N^{th} Fibonacci number (c) Length of a string (d) All
- Recursion is similar to which of the following?
(a) Switch Case (b) Loop (c) If-else (d) None
- In recursion, the condition for which the function will stop calling itself is:
(a) Best case (b) Worst case
(c) Base case (d) There is no such condition
- Consider the following code:

```
void my_recursive_function()
{
    my_recursive_function();
}
void main()
{
    my_recursive_function();
}
```

What will happen when the above code is executed?

- The code will be executed successfully and no output will be generated
- The code will be executed successfully and random output will be generated

- (c) The code will show a compile-time error
 (d) The code will run for some time and stop when the stack overflows
 Consider the following code:

```
void my_recursive_function(int n)
{
    if(n == 0)
        return;
    cout<< n<< " ";
    my_recursive_function(n-1);
}

void main()
{
    my_recursive_function(10);
}
```

What will be the output of the above code?

- (a) 10 (b) 1 (c) 10 9 8 ... 1 0 (d) 10 9 8 ... 1

What is the Base case for the following code?

```
void my_recursive_function(int n)
{
    if(n == 0)
        return;
    cout<< n<< " ";
    my_recursive_function(n-1);
}

void main()
{
    my_recursive_function(10);
}
```

- (a) return (b) cout<<n<< " ";
 (c) if(n == 0) (d) my_recursive_function(n-1)

How many times is the recursive function called, when the following code is executed?

```
void my_recursive_function(int n)
{
    if(n == 0)
        return;
    cout<<n<< " ";
    my_recursive_function(n-1);
}

void main()
{
    my_recursive_function(10);
}
```

- (a) 9 (b) 10 (c) 11 (d) 12

Which of the following statements is true?

- (a) Recursion is always better than iteration
 (b) Recursion uses more memory compared to iteration
 (c) Recursion uses less memory compared to iteration
 (d) Iteration is always better and simpler than recursion

15. What will be the output of the following code?

```
#include <iostream.h>
int fun(int n)
{
    if (n == 4)
        return n;
    else return 2*fun(n+1);
}
void main()
{
    cout<<fun(2);
}
```

- (a) 4 (b) 8 (c) 16 (d) Runtime error
16. Iteration uses a repetition structure whereas recursion uses:
 (a) Sorting structure (b) Selection structure (c) Controlling structure (d) All
17. Tower of Hanoi is a classic example of:
 (a) Divide and conquer (b) Recursive approach (c) b but not a (d) Both a & b
18. Recursion uses more memory space than iteration because:
 (a) It uses stack instead of a queue. (b) Every recursive call has to be stored.
 (c) Both a & b (d) None of these
19. If there is no Base criteria in a recursive function, the program will:
 (a) not be executed. (b) be executed until all conditions match
 (c) be executed infinitely. (d) be obtained progressive approach
20. Which data structure is used to perform recursion?
 (a) Queue (b) Stack (c) Linked List (d) Tree

Answers of MCQs

01 (b)	02 (c)	03 (a)	04 (b)	05 (c)	06 (c)	07 (d)	08 (b)	09 (c)	10 (d)
11 (d)	12 (c)	13 (c)	14 (b)	15 (c)	16 (b)	17 (d)	18 (b)	19 (c)	20 (b)

- Q.2 Describe recursion and its basic conditions. Moreover, write a program in C++ using a recursive function to display the first 10 even numbers.
- Q.3 Describe tail and non-tail recursions with examples.
- Q.4 What do you know about excessive recursion?
- Q.5 What are the advantages and disadvantages of recursion?
- Q.6 Write a program that inputs 10 integer values and pushes them into a stack. It pops values from the stack one by one, calculates the factorial of each using a recursive function, and displays the results on the screen.

Source Code of Programs

Source code of programs given in "DATA STRUCTURES USING C++" can be received by sending an e-mail to "pakman_series@hotmail.com".

QUEUES

QUEUES

In general terminology, a queue is a "waiting line" such as a line of people in front of a bank to deposit utility bills. Similarly, a queue of jobs in a computer system waiting for some service such as a printer or CPU to be processed further. In each of these examples, the objects or jobs are serviced in the order in which they arrive; that is, the first item in the queue is the first to be served. Following figures show some more examples of queues in real life:

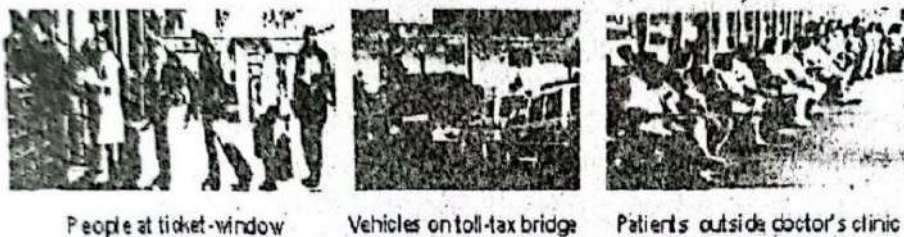


Figure 7.1: Examples of queues in real life

- ★ **A queue of people at the ticket-window:** The person who comes first, gets the ticket first. A person who wishes to keep standing in the line must go to the back of the line or queue. The person who comes last gets the ticket at last.
- ★ **Vehicles on the toll-tax bridge:** The vehicle that arrives first at the toll tax booth, leaves the booth first. The vehicle that comes last leaves last.
- ★ **Patients waiting outside the doctor's clinic:** The patient who comes first visits the doctor first; and the patient who comes last visits the doctor last.

7.1 Queue in Programming

In computer programming terminology, a queue is a linear data structure in which new items are inserted from one end, whereas existing items are removed from the other end. The end at which the items are inserted is called *Rear* or *Back* or *Tail* of the queue. The end from where items are removed is called *Front* or *Head* of the queue.

The item that is added at first into the queue is removed first. Similarly, the item that is added at the end is removed at the end. For this reason, a queue is also known as *First-In, First-Out* (FIFO) data structure.



Figure 7.2: Queue with 9 items

Figure 7.2 shows a queue with 9 items. The item ① is the first item, while ⑨ is the last item. A new item can only be added after ⑨. Similarly, the items can be removed starting at the front of the queue. For example, the first item to be removed will be ① and to remove the item ⑨, all items in front of ⑨ i.e. ①, ②, ③, ④, ⑤, ⑥, ⑦ and ⑧ must be removed first.

Difference between Stack and Queue

The major differences between stack and queue are as follows:

- The queue is open at both of its ends, one end is used to insert data items and the other is used to remove data items. Whereas stack is open at only one end which is called *Top*, the data items are added and removed at this end only.
- In a queue, data items are removed in the same order they were entered. In a stack, data items are removed in the reverse order from which they were entered.
- The queue is a First-In-First-Out (FIFO) data structure, while the stack is a Last-In-First-Out (LIFO) data structure.

7.1.2 Applications of Queues in Computer System

Queues have many applications in computer systems. Most computers have only a single processor, so the job of only one user can be performed at a time. The jobs of other users are placed in a queue and are performed one after the other. For example, if users want to print documents on the printer, the operating system (or a special print spooler) queues the documents by placing them in a special area called a *print buffer* or *print queue*. The document sent first is printed first and so on. Similarly, a network operating system uses queue to process printing requests. The documents are sent to the printer in a network environment (which may have a single printer). The printer prints the documents in the order in which they are received.

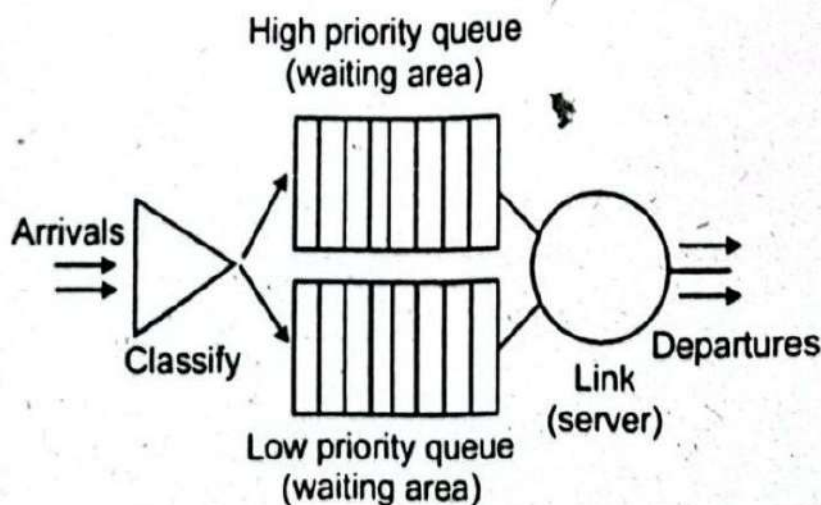


Figure 7.3: Priority queues in multiprogramming operating systems

In multiprogramming operating systems, where different users have various levels of priorities over the others, multiple queues are maintained to keep track of multiple jobs of these users. Jobs from these users arrive at the operating system randomly. These random jobs are classified into low priority and high priority jobs. A job with higher priority is placed in high priority queue whereas jobs with lower priority are kept in low priority queue as shown in figure 1.3. High priority jobs are departed first for processing, in the sequence in which they had arrived. Then low priority jobs are departed for processing in the same way.

In electronic data communication systems, queues are also used to receive and send information packets. The information packets wait in queues in a computer network to be moved to the next nodes.

1.1.3 Implementation of Queue

A queue can be implemented in computer programming languages using different ways. Normally, linear arrays and linked lists are used to implement queues. In this chapter, the queue is implemented using a linear array.

When an array is used to implement a queue, two variables called *back* and *front* are required. The variable *back* is used as a pointer to point the location where an item is inserted in an array. The variable *front* is used as a pointer to point the location where an item is removed from the array. Initially, both *front* and *back* are set to -1 (or when a queue is empty both *front* and *back* are set to -1). In this chapter, names for *back* and *front* pointers are used as *B* and *F* respectively. A queue is comparatively difficult to implement than that of a stack because two data pointers, *front*, and *back* have to be maintained in programming.

Note: Implementation of queues using a linked list is discussed in chapter 9 part-3.

1.1.4 Basic Features of Queue

Following are the basic features of the queue:

- 1- Like stack, the queue is also an ordered list of items or elements with similar data types.
- 2- The queue is a FIFO data structure (i.e. items are accessed in First In First Out) manner.
- 3- Once a new item is inserted into the queue, all the items inserted before the new item in the queue must be removed, to remove the new item.
- 4- Like a stack, a queue is also said to be in *OVERFLOW* state when it is completely full, whereas it is said to be in *UNDERFLOW* state when it is empty.

1.1.5 Queue Operations

There are two basic operations that are mostly performed on a queue. These operations are called *enqueue* and *dequeue*. A brief description of these operations is given as under:

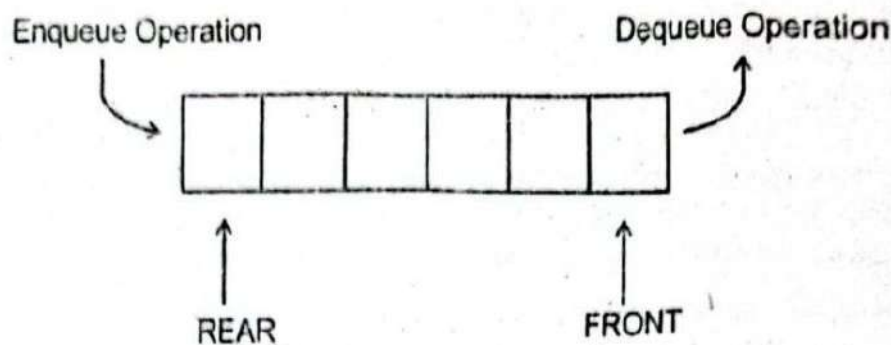


Figure 7.4: Queue Operations

7.1.5.1 Enqueue Operation

The process to add or insert an item into the queue is called *enqueue*. This operation always takes place at the back (rear) end of the queue.

The *enqueue* operation involves the following steps:

- 1 – Check if the queue is full.
- 2 – If the queue is full, produce an error message “overflow” and exit.
- 3 – If the queue is not full, make an increment to the *back* pointer to point to the empty space.
- 4 – Add the new data item to the queue location, where the *back* is pointing.
- 5 – Exit the operation.

7.1.5.2 Dequeue Operation

The process to remove (or delete) an item from the queue is called *dequeue*. This operation always takes place at the front end of the queue.

The *dequeue* operation involves the following steps:

- 1 – Check if the queue is empty.
- 2 – If the queue is empty, produce an error message “underflow” and exit.
- 3 – If the queue is not empty, access the data item to where the *front* is pointing.
- 4 – Increment *front* pointer to point to the next available data item in the queue.
- 5 – Exit the operation.

Example:

Suppose an array QU represents a queue and the variables F and B represent the *front* and *back* pointers of the queue respectively. This array has a capacity of storing '5' elements. Initially, both F and B are set to -1. Assign 0 to F if you have just inserted the first value into the queue.

- If a new data item is added into the array, the value of B is increased by 1 (firstly, the value of B is increased and then the value is inserted).
- If a data item is removed from the queue, the value of F is increased by 1.
- If the values of F and B are equal and the queue is not empty, then the queue has only one item.
- If the value of F is -1, then the queue is empty.
- If the value of B is greater than or equal to 5, then the queue is overflowed.

Following table illustrates the *enqueue* and *dequeue* operations on the array QU:

Queue Status	Operation
F B ↓ ↓ X 0 1 2 3 4	Add an item in the queue such as 'X'. The value of B is increased by 1 i.e. it becomes 0. Assign value 0 to F; because the first value is inserted into the queue. Therefore, both F and B point to position 0 of the array.
F B ↓ ↓ X Y 0 1 2 3 4	Add another item in the queue such as 'Y'. The value of B is increased by 1 i.e. it becomes 1. So B points to position 1 of the array.
F B ↓ ↓ X Y Z 0 1 2 3 4	Add another item in the queue such as 'Z'. The value of B is increased by 1 i.e. it becomes 2. So B points to position 2 of the array.
F B ↓ ↓ Y Z 0 1 2 3 4	Delete the front item from the queue which is 'X'. The value of F is increased by 1 i.e. it becomes 1. So F points to position 1 of the array.
F B ↓ ↓ Z 0 1 2 3 4	Delete another item from the front of the queue which is 'Y'. The value of F is increased by 1 i.e. it becomes 2. Both the values of F and B are equal. It means that only one item exists in the queue.
B F ↓ ↓ 0 1 2 3 4	Delete another item from the queue i.e. the last item which is 'Z'. The value of F is increased by 1 i.e. it becomes 3. At this position, the value of F is greater than B, so initialize the value of F & B to -1.

Algorithm – Inserting Data Item into Queue

Write an algorithm to insert an item into a queue 'QU' which has a capacity of N elements.

1. START
2. ASSIGN value -1 to F and B
3. IF $B \geq N$ THEN [Check queue if it has space]
 - PRINT "Queue overflow"
 - RETURN
- ELSE
 - $B = B + 1$
 - INPUT value in X
 - $QU[B] = X$ [Insert value X into queue]
- END IF
4. [Assign 0 to F if you have just inserted the first value into queue]
 - IF $F = -1$ Then
 - $F = 0$
- END IF
5. STOP

Algorithm – Removing Data Item from Queue

Write an algorithm to remove an item from a queue 'QU' which has a capacity of N elements.

1. START
2. IF $F = -1$ THEN [Check queue if it is empty]
 - PRINT "Queue is empty"
 - RETURN
- END IF
3. PRINT $QU[F]$ [Display the item and then remove it from queue]
 - $QU[F] = \text{NULL}$
 - $F = F + 1$
4. [Initialize value of F & B to -1 if last item is removed from queue]
 - IF $F > B$ THEN
 - $B = -1$
 - $F = -1$
- END IF
5. STOP

7.1.5.3 Complexity Analysis of Enqueue and Dequeue

The enqueue operation of a queue data structure is an insertion operation. A new element is directly inserted at the back of queue. While dequeue operation of a queue data structure is a deletion operation. An item is directly removed from the front of queue.

The time complexity and space complexity of enqueue and dequeue algorithms of the queue are as follows:

Operation	Time Complexity		Space Complexity
	Average-Case	Worst-Case	Worst-Case
Enqueue (Insertion)	$\Theta(1)$	$O(1)$	$O(n)$
Dequeue (Deletion)	$\Theta(1)$	$O(1)$	$O(n)$

Program Code 7.1

// program that inserts and removes data items to and from a simple queue.

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
class queue
```

```
{
```

```
private:
```

```
int B, F;
```

```
int QU[10];
```

```
public:
```

```
// constructor that initializes variables B & F
```

```
queue(void) {B = -1; F = -1;}
```

```
void insert(int);
```

```
void remove(void);
```

```
void display(void);
```

```
};
```

```
void main(void)
```

```
{
```

```
queue obj;
```

```
int n, opt, loop = 1;
```

```
while(loop)
```

```
{
```

```
clrscr();
```

```
cout<<"1- Insert value into queue "<<endl;
```

```
cout<<"2- Remove value from queue "<<endl;
```

```
cout<<"3- Display values of queue "<<endl;
```

```
cout<<"4- Exit "<<endl;
```

```
cout<<"Enter your option [1-4] : ";
```



```

        cin>>opt;
        switch(opt)
        {
            case 1:
                cout<<"Enter value to insert : ";
                cin>>n;
                obj.insert(n);
                break;
            case 2:
                obj.remove();
                break;
            case 3:
                cout<<"Values in queue\n";
                obj.display();
                break;
            case 4:
                loop = 0;          break;
            default:
                cout<<"Invalid option";
        } // end of switch
    } // end of while
} // end of main()

```

// member function to insert value into queue

```

void queue::insert(int x)
{
    if(B >= 9)
    {
        cout<<"Queue overflow";
        getch();
        return;
    }
    else
    {
        B = B + 1;
        QU[B] = x;
    }
    if(F == -1)
        F = 0;
} // end of insert()

```

// member function to remove value from queue

```

void queue::remove(void)
{
    if(F == -1)
    {

```

```

        cout<<"Queue is empty";
        getch();
        return;
    }
    cout<<"Value "<<QU[F]<<" is removed "<<endl;
    getch();
    QU[F] = NULL;
    F = F + 1;
    if(F>B)
        F = B = -1;
} // end of remove()

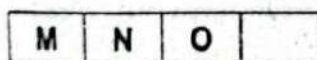
// member function to display values of queue
void queue::display(void)
{
    if(F == -1)
    {
        cout<<"Queue is empty";
        getch();
        return;
    }
    for(int i = F; i <= B; i++)
        cout<<QU[i]<<endl;
    getch();
} // end of display()

```

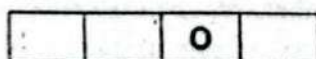
1.6 Drawback of Simple Queue

As discussed in the previous topic, in simple or normal queue data items are inserted at back end, while these are removed at the front end. However, a simple queue has a storage problem. A large amount of memory is required to store data items into this queue and to process these data items. Consider the following example to understand the problem.

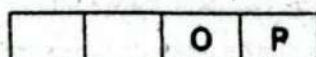
Consider a queue 'QU' that can hold a maximum of four items. Initially, the queue contains three items M, N, and O as shown below:



When items M and N are deleted, the queue will be as shown below:



In the above queue, the values of B and F are the same. Only one item can be added to the queue in this state. For example, when an item P is added, the queue will be as shown below:



At this stage, if another item such as 'Q' is to be added, it would have to be added after P, but there is no space after P. It shows that the queue is full and we cannot insert a new item into the queue. Although there are still two empty spaces in the queue, but adding another item into the queue would result in an overflow message.

When items are deleted from the front, their spaces cannot be used to store new items. This is the major problem in the normal queue data structure. A circular queue is used to overcome this problem (or limitation of the simple queue).

7.2 CIRCULAR QUEUE

A circular queue is a linear data structure. It follows the FIFO (First In First Out) principle. In a circular queue, the last node is connected back to the first node to make a circle. A circular queue is also called a *circular buffer*, *ring buffer*, or *cyclic buffer*. Please note that it is circular logically, not physically because the last node only points to the first node of the queue. A graphical representation of a circular queue is shown in figure 7.5.

A circular queue is very efficient; because insertion and deletion are totally independent of each other. We can use memory (in the circular queue) more efficiently.

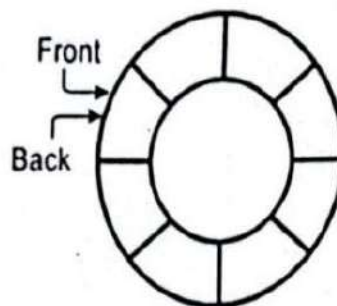


Figure 7.5: Circular Queue

7.2.1 Applications of Circular Queue

Circular queues have many applications in computer systems. Following are some applications of the circular queue:

- **Efficient Memory Management:** The unused memory locations can be utilized in a better way in circular queues.
- **Computer-Controlled Traffic System:** In a computer-controlled traffic system, circular queues are used to operate the traffic lights repeatedly. When one cycle of operation is completed, it again moves to the start of the cycle and thus constantly keeps on operating.
- **Burning a DVD:** When burning a DVD it is essential that the laser beam burning pits onto the surface is constantly fed with data, otherwise the DVD fails. Most leading DVD burn applications make use of a circular buffer to stream data from the hard disk onto the DVD. The first part i.e. the "writing process", fills up a circular buffer with data, then the "burning process" begins to read from the buffer as the laser beam burns pits onto the surface of the disk. If the buffer starts to become empty, the application carefully slows down the burn rate and speeds up the writing rate to avoid "buffer underflow" i.e. an empty buffer.

1.2.2 Implementation of Circular Queue

Like a simple queue, a circular queue can be implemented using a linear array or linked list. In this chapter, the linear array is used for the implementation of the circular queue. In linear array, the range of a subscript is '0' to 'n-1', where n is the maximum size of the array. Like a simple queue, in a circular queue, both *front* and *back* pointers are used. The *front* pointer is denoted by F, while the *back* pointer is denoted by B.

1.2.3 Circular Queue Operations

Like a simple queue, there are two basic operations that are performed in a circular queue. These operations are *insertion* (enqueue) and *deletion* (dequeue). A brief description of these operations is given as under:

1.2.3.1 Insertion Operation

Insertion operation is used to add (or insert) an item or element into the circular queue. The insertion operation involves the following steps:

- 1- Check if the circular queue is full. If it is full, produce an error message "overflow" and exit. A circular queue is declared as full if the value of F is equal to $(B+1) \% \text{MAX}$, where MAX indicates the maximum size of the array, and % is the division operator giving the remainder of a division.
- 2- If the circular queue is not full, set the value of B to point to the next empty space and then insert new data item into the circular queue. In insertion operation, the value of B is set or updated using the following formula:

$$B = (B+1) \% \text{MAX}$$

- 3- Check if the value of F is -1; if true then set the value of F to 0.
- 4- Exit the operation.

1.2.3.2 Deletion Operation

Deletion operation is used to remove (or delete) an item from a circular queue. The deletion operation involves the following steps:

- 1- Check if the circular queue is empty. If it is empty, produce an error message "underflow" and exit. A circular queue is declared as empty if both F and B are equal to -1.
- 2- If the queue is not empty, delete the data item from the location to where F is pointing.
- 3- If the value of B is equal to F, then set the values of both F and B to -1, otherwise set the value of F using the following formula:

$$F = (F+1) \% \text{MAX}$$

- 4- Exit the operation

Example:

Figure 7.6 shows circular queues with different data items. The maximum size of the array is 5. Figure (1) shows a circular queue with three numbers 20, 5, and 42. The index value of F is 0 and B is 2.

Figure (2) shows a circular queue after deleting numbers 20 and 5. It contains only one data item. The index value of F and B is 2. It means that if the circular queue contains a single data item, F is equal to B. After removing the last data item, initialize F and B to -1. When both F and B are equal to -1, the queue is declared as empty. Figure (3) shows an empty circular queue.

Figures (4), (5), and (6) show full circular queues with different values of F and B. A circular queue is declared as full queue if the value of F is equal to $(B+1) \% \text{MAX}$ (MAX means the maximum size of an array).

In figure (4), value of B is 4, F is calculated as: $4+1 \% 5 = 0$.

In figure (5), value of B is 1, F is calculated as: $1+1 \% 5 = 2$.

In figure (6), value of B is 3, F is calculated as: $3+1 \% 5 = 4$.

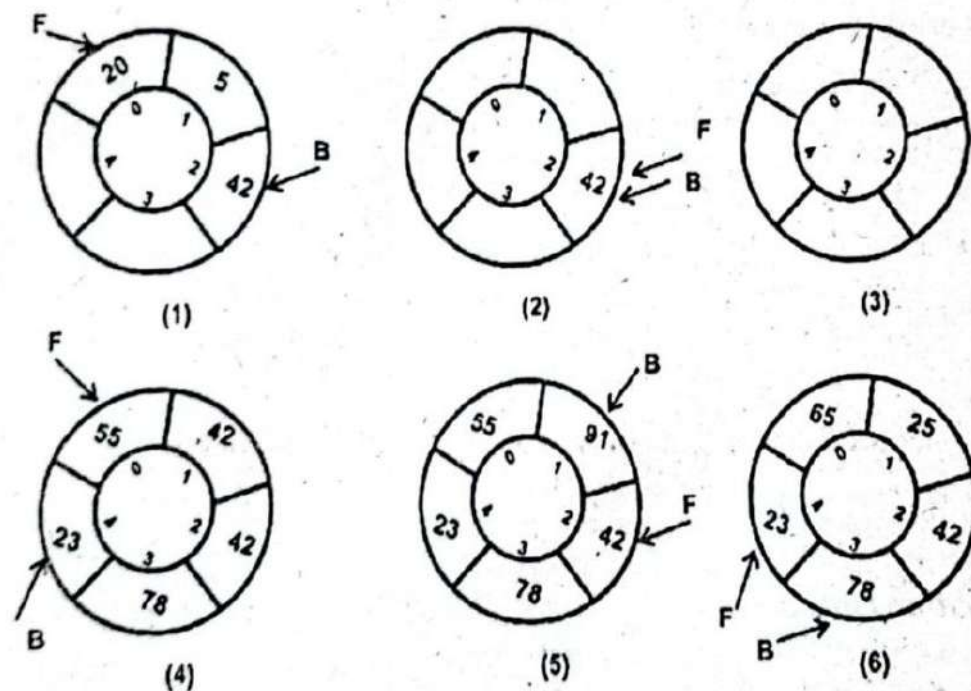


Figure 7.6: Circular Queues

Function : Insert(X)

Write a function sub-algorithm to Insert an Item 'X' into a circular queue 'CQ'. F and B represent the pointers to the Front and Back of the queue respectively. MAX represents the maximum size of the queue.

INSERT(X)

1. IF $F = (B + 1) \% \text{MAX}$

PRINT "Circular Queue Overflow"

```

        RETURN
    ELSE
        B = (B + 1) % MAX
        CQ[B] = X
    END IF
2.   IF F = -1 THEN F = 0
3.   RETURN

```

Function : Delete()

Write a function sub-algorithm to delete an item 'X' from a queue 'CQ'. F and B represent the pointers to the Front and Back of the queue. The deleted item is assigned to variable X. MAX represents the maximum size of the queue.

X = DELETE()

```

1.   IF (F = -1) AND (B = -1)
        PRINT "Circular Queue Empty"
        RETURN
    ELSE
        X = CQ[F]
        CQ[F] = NULL
    END IF
2.   IF F = B THEN
        F = -1
        B = -1
    ELSE
        F = (F + 1) % MAX
    END IF
3.   RETURN X

```

Complexity Analysis of Circular Queue

The time complexity and space complexity of circular queue operations are similar to linear operations:

Operation	Time Complexity		Space Complexity
	Average-Case	Worst-Case	Worst-Case
Insertion	$\Theta(1)$	$O(1)$	$O(n)$
Deletion	$\Theta(1)$	$O(1)$	$O(n)$

Program Code 7.2

// program to insert and delete items to and from a circular queue.

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
class cir_que
```

```
{
```

```
private:
```

```
int F, B;
```

```
int CQ[5];
```

```
public:
```

```
// constructor to initialize F & B
```

```
cir_que()
```

```
{
```

```
F = -1;
```

```
B = -1;
```

```
}
```

```
void qinsert(int);
```

```
int qdel(void);
```

```
};
```

```
void main(void)
```

```
{
```

```
    cir_que obj;
```

```
    int opt, val;
```

```
    while(opt!=3)
```

```
    {
```

```
        clrscr();
```

```
        cout<<"1 : Insert Item\n";
```

```
        cout<<"2 : Delete Item\n";
```

```
        cout<<"3 : Exit\n";
```

```
        cout<<"Enter your choice : ";
```

```
        cin>>opt;
```

```
        switch(opt)
```

```
        {
```

```
            case 1:
```

```
                cout<<"Enter value to insert : ";
```

```
                cin>>val;
```

```
                obj.qinsert(val);
```

```
                break;
```

```
            case 2:
```

```
                cout<<"\nValue "<<obj.qdel()<<" is deleted\n";
```

```
        } // end of switch
```

```
    } // end of while
```

```
} // end of main()
```

```
//member function to add item
void cir_que::qinsert(int n)
{
    if(F == (B+1)%5)           // maximum size of array is 5
    {
        cout<<"Circular Queue is full";
        getch();
        return;
    }
    else
    {
        B = (B + 1) % 5;
        CQ[B] = n;
        cout<<"\n Value "<<n<<" is inserted\n";
        getch();
    }
    if(F == -1)
        F = 0;
} // end of qinsert()
```

```
//member function to delete item
int cir_que::qdel()
{
    int x;
    if((F == -1)&&(B == -1))
    {
        cout<<"Circular Queue is empty";
        getch();
        return NULL;
    }
    else
    {
        x = CQ[F];
        CQ[F] = NULL;
    }
    if(F == B)
        F = B = -1;
    else
        F = (F + 1) % 5;
    return x;
} // end of qdel()
```


7.3 DEQUE

The term deque stands for *double-ended queue* (deque is pronounced as "deck"). Deque is a linear data structure in which items can be added and removed at both ends (*front* and *back*). However, an item cannot be added or deleted in the middle of the deque. A deque with some values is shown in the following figure:

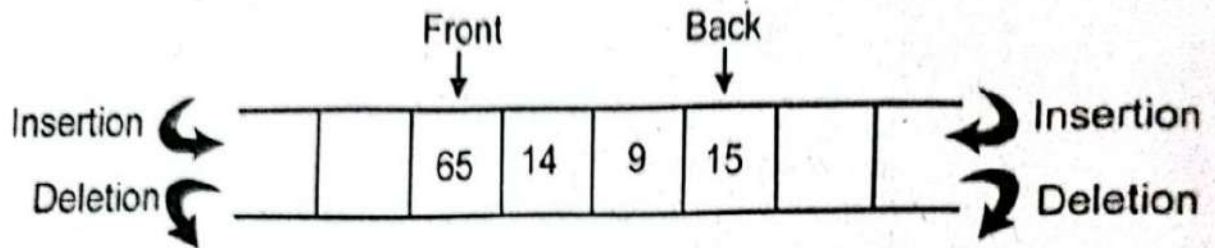


Figure 7.7: Deque

Types of Deques

There are two types of deques:-

1. **Input-Restricted Deque:** This type of deque allows insertion only at one end but allows deletion at both ends.
2. **Output-Restricted Deque:** This type of deque allows deletion only at one end but allows insertion at both ends.

7.3.1 Applications of Deque

Deque has many applications in real-world situations and computer systems. Some applications are as follows:

- In the air ticket purchasing line, the deque is like a queue. But sometimes, it happens that somebody has purchased the ticket but suddenly he/she comes back to have some query, in front of the queue. In this case, because the person has already purchased the ticket therefore he/she should be given a privilege (right) in case of any confusion or query. So in this kind of situation, we need a data structure where according to requirement we can add data from the front, as well.
- In a computer system, a good application of the deque is storing a web browser's history of web pages. Most recently visited URLs are added to the front of the deque, and the URL at the back of the deque is removed after some specified number of insertions at the front.

7.3.2 Implementation of Deque

Deque can be implemented in computer languages in different ways. Usually, it is implemented in a similar way as a circular queue is implemented i.e. linear array and linked list. In this chapter, the deque is implemented by using a linear array with two pointer variables pointing to its two ends.

7.3.3 Deque Operations

There are four basic operations that are performed on a deque. These operations are as follows:

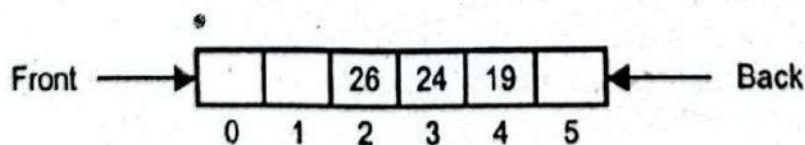
- 1) Insertion at Front
- 2) Insertion at Back
- 3) Deletion from Front
- 4) Deletion from Back

The following points must be noted about the deque operations (Suppose F represents front and B represents back of deque):

- ★ When an item is added at the front of the deque, then F is decreased by 1.
- ★ When an item is added at the back of the deque, then B is increased by 1.
- ★ When an item is removed from the front of the deque, the value of F is increased by 1.
- ★ When an item is removed from the back of the deque, the value of B is decreased by 1.
- ★ If the values of F and B are equal, it indicates that the deque has only one item.
- ★ After deleting the last item in the deque, the values of both F and B are set to -1 (or NULL value). It indicates that the deque is empty.

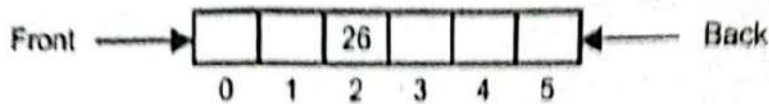
Example:

Following figure of deque shows some items stored in it:



The values of F and B for the above deque are 2 and 4 respectively.

If two values 19 and 24 are removed from the back of the deque, then B decreases by 2. So the value of B becomes 2. The values of F and B are equal. It indicates that the deque has only one item which is 26.



Algorithm – Insertion at the front side of the deque

Write an algorithm to Insert an item at the front end of a deque **DQ** having elements.

- 1- START
- 2- INPUT data value into X
- 3- IF $F = 0$ AND $B = N-1$ THEN [When deque is full]
 PRINT "Deque is full"
 EXIT
 END IF
- 4- IF $F = -1$ AND $B = -1$ THEN [When deque is empty]
 $B = F = 0$
 $DQ[F] = X$
 END IF
- 5- IF $F > 0$ THEN [When deque has space at its front end]
 $F = F - 1$
 $DQ[F] = X$
 ELSE
 PRINT "No space at front side of the deque"
 END IF
- 6- END

Algorithm – Insertion at the back side of the deque

Write an algorithm to Insert an item at the back (rear) end of a deque **DQ** having elements.

- 1- START
- 2- INPUT data value into X
- 3- IF $F = 0$ AND $B = N-1$ THEN [When deque is full]
 PRINT "Deque is full"
 EXIT
 END IF
- 4- IF $F = -1$ AND $B = -1$ THEN [When deque is empty]
 $B = F = 0$
 $DQ[B] = X$
 END IF
- 5- IF $B < N-1$ THEN [When deque has space at its back end]

```

        B = B + 1
        DQ[B] = X
    ELSE
        PRINT "No space at back side of the deque"
    END IF
6-    END

```

Algorithm – Deletion from the front side of the deque

Write an algorithm to delete an item from the front side of a deque which consists of N elements.

```

1-    START
2-    IF F = -1 AND B = -1 THEN    [When deque is empty ]
        PRINT "Deque is empty"
        EXIT
    ELSE                            [When deque is not empty ]
        DQ[F] = NULL
    END IF
3-    [After deleting front value, check values of F and B and set their values]
    IF F = B THEN
        B = F = -1
    ELSE IF F = N-1 THEN
        F = -1
    ELSE
        F = F + 1
    END IF
4-    END

```

Algorithm – Deletion from the back (rear) side of the deque

Write an algorithm to delete an item from the backside of a deque which consists of N elements.

```

1-    START
2-    IF F = -1 AND B = -1 THEN    [When deque is empty ]
        PRINT "Deque is empty"
        EXIT
    ELSE                            [When deque is not empty ]
        DQ[B] = NULL
    END IF

```


- 3- [After deletion back value, check values of F and B and set their values]
 IF B = F THEN
 B = F = -1
 ELSE
 B = B - 1
 END IF
 4- END

Complexity Analysis of Deque

The time complexity and space complexity of deque operations are similar to queue, circular queue operations:

Operation	Time Complexity		Space Complexity
	Average-Case	Worst-Case	Worst-Case
Insertion	$O(1)$	$O(1)$	$O(n)$
Deletion	$O(1)$	$O(1)$	$O(n)$

Program Code 7.3

```
// program to insert and delete items from a deque
#include <iostream.h>
#include <conio.h>

class deque
{
    private:
        int F, B, DQ[5];
    public:
        // constructor to initialize F & B
        deque()
        {
            F = -1; B = -1;
        }
        void insert_front(int);
        void insert_back(int);
        void del_front(void);
        void del_back(void);
        void print_deque(void);
};
```

```
void main(void)
{
    deque obj;
    int opt, val;
    while(opt!=5)
    {
        clrscr();
        cout<<"1 : Insert Item from Front\n";
        cout<<"2 : Insert Item from Back\n";
        cout<<"3 : Delete item from Front\n";
        cout<<"4 : Delete Item from Back\n";
        cout<<"5 : Exit\n";
        cout<<"Enter your choice :{1-5} : ";
        cin>>opt;
        switch(opt)
        {
            case 1:
                cout<<"\nEnter value to insert : ";
                cin>>val;
                obj.insert_front(val);
                obj.print_deque();
                break;

            case 2:
                cout<<"\nEnter value to insert : ";
                cin>>val;
                obj.insert_back(val);
                obj.print_deque();
                break;

            case 3:
                obj.del_front();
                obj.print_deque();
                break;

            case 4:
                obj.del_back();

        } // end of switch
    } // end of while
} // end of main()

//member function to insert item from front side of the deque
void deque::insert_front(int n)
{
```



```
        if(F == 0 && B == 4 )
        {
            cout<<"Deque is full";
            return;
            getch();
        }
        if(F == -1 && B == -1)
        {
            B = F = 0;
            DQ[F] = n;
        }
        else if(F > 0)
        {
            F--;
            DQ[F] = n;
        }
        else
        {
            cout<<"No space from front side ";
            getch();
        }
    } // end of insert_front()

    // member function to insert item from back side of the deque
    void deque::insert_back(int n)
    {
        if(F == 0 && B == 4)
        {
            cout<<"Deque is full";
            return;
            getch();
        }
        if(F == -1 && B == -1)
        {
            B = F = 0;
            DQ[B] = n;
        }
        else if(B < 4)
        {
            B++;
            DQ[B] = n;
        }
        else
```

```

    {
        cout<<"No space from rear side ";
        getch();
    }
} // end of insert_back()

// member function to delete item from front side of the deque
void deque::del_front()
{
    if(F == -1 && B == -1)
    {
        cout<<"Deque is empty";
        getch();
        return;
    }
    else
        DQ[F] = NULL;
    if(F == B)
        F = B = -1;
    else if(F == 4)
        F == -1;
    else
        F++;
} // end of del_front()

// member function to delete item from back side of the deque
void deque::del_back()
{
    if(F == -1 && B == -1)
    {
        cout<<"Deque is empty";
        getch();
        return;
    }
    else
        DQ[B] = NULL;
    if(B == F) F = B = -1;
    else B--;
} // end of del_back()

// member function to display data from deque
void deque::print_deque()
{
    cout<<"\nDeque after operation\n";
}

```



```

    if(F == -1)
    {
        cout<<"Queue is empty";
        return;
        getch();
    }
    for(int i = F; i<=B; i++)
        cout<<DQ[i]<< " ";
    getch();
} // end of print()

```

7.4 PRIORITY QUEUES

The items in queues are removed in the order in which they are added. For example, when jobs are sent to the printer, they are placed in a queue. The job sent first is printed first and the job sent in the end is printed at the end. However, it may not always fulfill the requirements of the real world. Some jobs may be more important than others and the user may like to print important jobs on priority basis. These types of jobs should have a preference also in the queue over the other jobs. For this purpose, a special type of queue, known as a *priority queue*, is used.

A priority queue is a data structure of items in which each item is assigned a priority. The items in the priority queue are processed according to the following rules:

- ★ The item with higher priority is served (or processed) before items with lower priority though they had arrived earlier in the queue. The item with the highest priority will be moved to the front of the queue and the one with the lowest priority will move to the back of the queue. Thus when an item is inserted at the back in the queue, it can move to the front because of its higher priority.
- ★ If two or more items have the same priority, these are served (or processed) according to the order in which they were added to the queue.

The following figure shows a priority queue. It represents jobs waiting to be processed on the computer. The priority numbers 1, 2, 3, and so on are attached to each job, where a lower number indicates higher priority, and higher numbers indicate lower priority.

Front	A	B	X	Y	C	D	Q	R	W	Back
Priority	1	1	2	3	2	2	3	3	3	

The priority number 1 is given to jobs A and B, priority number 2 is given to jobs X, C, and D, and priority number 3 is given to the jobs Y, Q, R, and W.

The insertion and deletion in a priority queue are based on the priority of the items. These can take place at any position within the queue. The queue shown in the above figure is a single queue in which insertion can be done at any position. This priority queue can be thought of as a combination of three queues, based upon the priorities of items, as shown below:

Priority 1	A	B				Queue-1
Priority 2	X	C	D			Queue-2
Priority 3	Y	Q	R	W		Queue-3

The items in Queue-2 can be removed only when the Queue-1 is empty. Similarly, to remove items from Queue-3, both the Queue-1 and Queue-2 must be empty. Similarly, when a new item is to be inserted, its priority is examined. Priority number 1 items are inserted into Queue-1, priority number 2 items into Queue-2, and priority number 3 items into Queue-3.

4.1 Application of Priority Queues In Operating Systems

An important application of priority queues in almost all operating systems is shown in figure 7.8.

We know that different types of processes or tasks continue to create in the main memory of a computer system. Some of these processes are more important than the other. For example, system processes (or internal tasks of OS) have the highest priority and need to be executed as soon as possible. Therefore, these processes are kept in the highest priority queue where they are executed normally in First Come First Served (FCFS) fashion. Next are shown interactive processes (needing input from users) and interactive editing (needing input along with editing from users) processes. Therefore, interactive processes are placed in the second priority queue and interactive editing processes in the third priority queue, according to their nature of working. Batch processes are the longest jobs and therefore, they are placed in the fourth priority queue. In the last priority queue, student processes are placed which do not need a quick response from the operating system. Please note that in these priority queues different scheduling algorithms like First Come First Served (FCFS), Shortest Job First (SJF), or Round Robin (RR) can be applied.

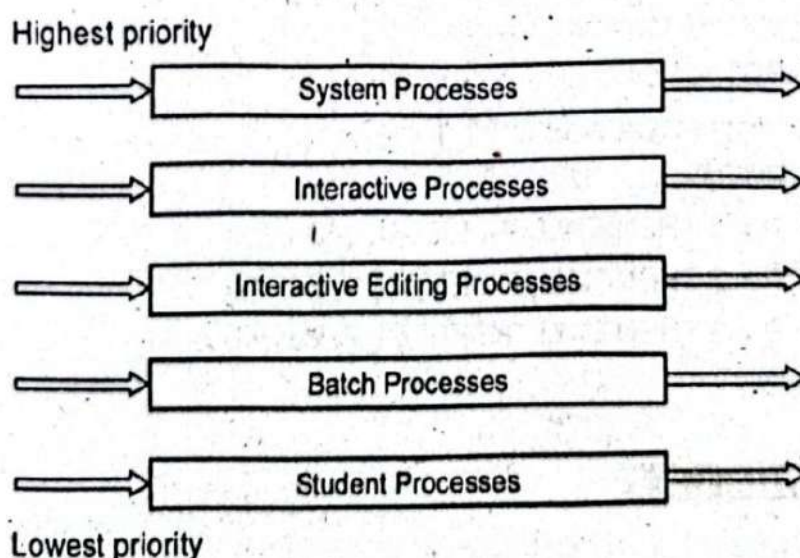


Figure 7.8: Application of priority queues in operating systems

7.4.1 Implementation of Priority Queue

The priority queues can be implemented in memory in several ways. The best way is to use a separate queue for each level of priority. Each such queue is represented in a circular fashion and has its own front and back pointers.

Usually, an array of arrays i.e. a two-dimensional array is used to implement the priority queue. Following figure shows the implementation of the priority queue:

1	A	B		
2	X	C	D	
3	Y	Q	R	W
4	--	--	--	

Algorithm – Insertion in Priority Queue

Write an algorithm to insert items into the priority queue represented by a two-dimensional array with 3 rows and 10 columns. Assume that the name of the array is ABC.

- 1- START
- 2- [initialize Back & Front for first row of two-dimensional array]
SET B1 & F1 to -1
- 3- [initialize Back & Front for 2nd row of two-dimensional array]
SET B2 & F2 to -1
- 4- [initialize Back & Front for 3rd row of two-dimensional array]
SET B3 & F3 to -1
- 5- [Enter value to insert, in variable 'N']
INPUT N
- 6- [Enter priority in variable 'P']
INPUT P
- 7- IF P = 1 THEN CALL INSERT(N, 0, B1, F1)
- 8- IF P = 2 THEN CALL INSERT(N, 1, B2, F2)
- 9- IF P = 3 THEN CALL INSERT(N, 2, B3, F3)
- 10- END

Function: INSERT(X, I, B, F)

- 1- IF B >= 9 THEN [Check queue whether it has space]
PRINT "Queue is Full"

```

        RETURN
    ELSE
        B = B + 1
        ABC[I][B] = X
    END IF
2-    IF F = -1 THEN F = 0

```

Algorithm – Deletion in Priority Queue

Write an algorithm to delete items from a priority queue consisting of a two-dimensional array "ABC" with 3 rows and 10 columns.

```

1-    START
2-    INPUT P      [Enter priority in variable 'P']
3-    IF P = 1 THEN X = DELETE(0, B1, F1)
4-    IF P = 2 THEN X = DELETE(1, B2, F2)
5-    IF P = 3 THEN X = DELETE(2, B3, F3)
6-    PRINT X, "is deleted from the queue"
7-    END

```

Function: Delete(B, F, I)

```

1-    IF F = -1 THEN
        PRINT "Queue is Empty"
        RETURN NULL
    END IF
2-    DATA = ABC[I][F]
3-    ABC[I][F] = NULL
4-    IF F = B THEN
        F = B - 1
    ELSE
        F = F + 1
    END IF
5-    RETURN DATA

```

Program Code 7.4

```

// program to insert and/or delete items from priority queue
// having 3 arrays with 10 elements in each array

#include <iostream.h>
#include <conio.h>

```



```

class pri_que
{
    private:
        int F1, B1, F2, B2, F3, B3;
        int PQ[3][10];
    public:
        pri_que() // constructor to initialize Fronts & Backs
        {
            F1 = -1; B1 = -1;
            F2 = -1; B2 = -1;
            F3 = -1; B3 = -1;
        }
        void insert(int, int);
        void add(int, int&, int&, int);
        del (int);
        int remove(int&, int&, int);
        void print(int);
        void ppp(int&, int&, int);
};

void main(void)
{
    pri_que obj;
    int opt, val, pro;
    while(opt!=3)
    {
        clrscr();
        cout<<"1 : Insert Item\n";
        cout<<"2 : Delete Item\n";
        cout<<"3 : Exit\n";
        cout<<"Enter the choice:[1-3] : ";
        cin>>opt;
        switch(opt)
        {
            case 1:
                cout<<"Enter Value to insert : ";
                cin>>val;
                cout<<"Enter Priority : ";
                cin>>pro;
                obj.insert(val, pro);
                cout<<"\nQueue after insertion\n";
                obj.print(pro);
                break;

```

```
        case 2:
            cout<<"Enter priority : ";
            cin>>pro;
            cout<<"\n Value "<<obj.del(pro) <<" is deleted\n";
            cout<<"\n Queue after deletion\n";
            obj.print(pro);
            getch();
    } // end of switch()
} // end of while loop
} // end of main()

// member function that calls add() function for inserting value in a queue
void pri_que::insert(int n, int p)
{
    switch(p)
    {
        case 1:
            add(n, B1, F1, 0);
            break;
        case 2:
            add(n, B2, F2, 1);
            break;
        case 3:
            add(n, B2, F2, 2);
            break;
    }
} // end of insert()

// member function add() that inserts value in a queue
void pri_que::add(int x, int& B, int& F, int i)
{
    if(B>=9)
    {
        cout<<"Queue is full";
        getch();
        return;
    }
    else
    {
        B++;
        PQ[i][B] = x;
        if(F == -1 ) F = 0;
    }
}
```



```
} // end of add()
```

```
// member function that calls remove() function for deleting value from the queue
```

```
pri_que::del(int p)
```

```
{
```

```
    switch (p)
```

```
    {
```

```
        case 1:
```

```
            remove(B1, F1, 0);
```

```
            break;
```

```
        case 2:
```

```
            remove(B2, F2, 1);
```

```
            break;
```

```
        case 3:
```

```
            remove(B2, F2, 2);
```

```
            break;
```

```
    }
```

```
} // end of del()
```

```
// member function remove() that deletes a value from the queue
```

```
int pri_que::remove(int& B, int& F, int i)
```

```
{
```

```
    int data;
```

```
    if(F == -1)
```

```
    {
```

```
        cout<<" Queue is empty";
```

```
        getch();
```

```
        return NULL;
```

```
    }
```

```
    data = PQ[i][F];
```

```
    if(F == B) F = B = -1;
```

```
    else F++;
```

```
    return data;
```

```
} // end of remove()
```

```
// member function that calls ppp() function for printing values from the queue
```

```
void pri_que::print(int p)
```

```
{
```

```
    switch(p)
```

```
    {
```

```
        case 1:
```

```
            ppp(B1, F1, 0);
```

```
            break;
```

```

        case 2:
            ppp(B2, F2, 1);
            break;
        case 3:
            ppp(B2, F2, 2);
            break;
    }
} // end of print()

// member function ppp() that displays values from the queue
void pri_que::ppp(int& B, int& F, int i)
{
    if(F == -1)
    {
        cout<<"Queue is empty";
        return;
    }
    for(int c = F; c < B; c++)
        cout<<PQ[i][c]<< "\t";
} // end of ppp()

```

Source Code of Programs

Source code of programs given in "DATA STRUCTURES USING C++" can be received by sending an e-mail to "pakman_series@hotmail.com".

Summary --- Complexities of Queue

Operation	Time Complexity		Space Complexity
	Average-Case	Worst-Case	Worst-Case
Insertion	$\Theta(1)$	$O(1)$	$O(n)$
Deletion	$\Theta(1)$	$O(1)$	$O(n)$
Access	$\Theta(n)$	$O(n)$	$O(n)$
Search	$\Theta(n)$	$O(n)$	$O(n)$

Short Answers to the Questions



Briefly explain different types of queues.

The queues can be of four types:

1. **Simple Queue:** In a simple queue, insertion occurs at the back (rear) end, and deletion occurs at the front end.
2. **Circular Queue:** In a circular queue, the last node is connected back to the first node to make a circle.
3. **Deque (Double Ended Queue):** In deque, insertion, and deletion occur at both ends i.e. front and back of the queue.
4. **Priority Queue:** A priority queue is a queue that contains items that have some preset priority. When an element has to be removed from a priority queue, the item with the highest priority is removed first.



Differentiate between stack and queue.

Following table shows the differences between stack and queue:

Sr#	Stack	Queue
1	Data items are inserted and removed at the same end.	Data items are inserted and removed from different ends.
2	In a stack, only one pointer is used which points to the top of the stack.	In a queue, two different pointers are used for front and rear ends.
3	In a stack, the last inserted data item is the first to come out. Therefore, Stack follows the Last In First Out (LIFO) order.	In a queue, the data item inserted first is deleted first. Therefore, Queue follows the First In First Out (FIFO) order.
4	Stack operations are called <i>push</i> and <i>pop</i> .	Queue operations are called <i>enqueue</i> and <i>dequeue</i> .
5	A stack is visualized as vertical collections.	A queue is visualized as horizontal collections.
6	The collection of dinner plates at a wedding reception is an example of a stack.	People standing in a line outside a ticket-window for receiving tickets is an example of a queue.



What is the difference between deque and simple queue?

A deque is a double-ended queue, which allows insertion and deletion from either end. Queue only allows insertion at one end and deletion from the other.



What is the advantage of a circular queue over a simple one?

In a circular queue, we utilize memory efficiently because in a simple queue when we delete any element, the only front is incremented by 1, but that position is not used later.

So when we perform more insertion and deletion operations then memory is wasted. But in a circular queue, memory can be utilized after deletion of any element from the front, because it is circular.



Differentiate between deque and circular queue.

Following table shows the differences between deque and circular queue:

Sr#	Deque	Circular Queue
1	In a deque, write and read operations take place in both directions.	In a circular queue, write and read operations only take place in one direction.
2	Four basic operations are performed on a deque. These operations are insertion at the front, insertion at back, deletion from the front, and deletion from the back.	Two basic operations are performed on a circular queue. These operations are insertion (enqueue) and deletion (dequeue).

EXERCISE

Q#1 Select the correct answer from the given multiple choices.

- A line of vehicles on a petrol pump or CNG station is an example of:
 - FIFO
 - FILO
 - LIFO
 - None
- The end at which the items are inserted is called _____ of the queue.
 - Tail
 - Rear
 - Back
 - All
- The end from where items are removed is called _____ of the queue.
 - Front
 - Tail
 - Back
 - All
- The insert operation in queue is known as:
 - enqueue
 - dequeue
 - deque
 - None
- A type of deque which allows insertions only at one end, but allows deletions at both ends, is known as:
 - Input-Restricted Deque
 - Output-Restricted Deque
 - dequeue
 - None
- A type of deque which allows deletions only at one end but allows insertions at both ends, is known as:
 - Input-Restricted Deque
 - Output-Restricted Deque
 - dequeue
 - None
- A priority queue is usually implemented using:
 - linear array
 - stack
 - two-dimensional array
 - None

8. A linear data structure in which items can be added or removed at either end, is called:
 (a) Stack (b) Array (c) Queue (d) Deque
9. FIFO stands for:
 (a) First Insertion First Operation (b) First Insert First Out
 (c) First Input First Output (d) First In First Out
10. Which one queue is used in hospital emergency room, where patients with heart trouble or other serious condition need to be attended before patients having less serious problems?
 (a) Queue (b) Priority Queue (c) Deque (d) None
11. One difference between a queue and a stack is:
 (a) Queue requires dynamic memory, but stack does not.
 (b) Stack requires dynamic memory, but queue does not.
 (c) Queue uses two ends of the structure, stack uses only one.
 (d) Stack uses two ends of the structure, queue uses only one.
12. If the characters 'D', 'C', 'B' and 'A' are placed in a queue (in that order), and then removed one at a time, in what order will they be removed?
 (a) ABCD (b) ABDC (c) DCAB (d) DCBA
13. If the characters 'A', 'B', 'C', and 'D' are placed in a queue and are deleted one at a time, in what order will they be removed?
 (a) ABCD (b) DCBA (c) DCAB (d) DABC
14. Suppose top is called on a priority queue that has exactly two entries with equal priority. How is the return value of the top selected?
 (a) The implementation gets to choose either one.
 (b) The one which was inserted first.
 (c) The one which was inserted most recently.
 (d) This can never happen (violates the precondition)
15. A linear array of elements in which deletion can be done from one end (front) and insertion can take place only at the other end (rear) is known as:
 (a) Queue (b) Stack (c) Tree (d) Linked list
16. Suppose the following circular queue can accommodate maximum six elements with the following data: front = 2, rear = 4 and queue is __, L, M, N, __, __
 What will happen after ADD operation takes place?
 (a) front = 2 rear = 5 and queue is __, L, M, N, O, __
 (b) front = 3 rear = 5, and queue is __, __, L, M, N, O
 (c) front = 3 rear = 4, and queue is __, __, L, M, N, O
 (d) front = 2 rear = 4, and queue is L, M, N, O, __, __
17. A queue is a:
 (a) FIFO (First In First Out) list (b) LIFO (Last In First Out) list
 (c) Ordered Array (d) Linear Tree

18. If the `MAX_SIZE` is the size of the array used in the implementation of circular queue, how is `rear` manipulated while inserting an element in the queue?
- (a) $\text{Rear} = (\text{Rear} \% 1) + \text{MAX_SIZE}$ (b) $\text{Rear} = \text{Rear} \% (\text{MAX_SIZE} + 1)$
(c) $\text{Rear} = (\text{Rear} + 1) \% \text{MAX_SIZE}$ (d) $\text{Rear} = \text{Rear} + (1 \% \text{MAX_SIZE})$
19. Suppose `MAX_SIZE` is the size of the array used in the implementation of circular queue, array index starts with 0, `Front` points to the first element in the queue, and `Rear` points to the last element in the queue. Which of the following condition specifies that circular queue is FULL?
- (a) $\text{Front} = \text{Rear} = -1$ (b) $\text{Front} = (\text{Rear} + 1) \% \text{MAX_SIZE}$
(c) $\text{Rear} = \text{Front} + 1$ (d) $\text{Rear} = (\text{Front} + 1) \% \text{MAX_SIZE}$
20. Suppose a circular queue is implemented using an array of size 10. The array index starts with 0, the front is 6, and the rear is 9. The insertion of the next element takes place at the array index:
- (a) 0 (b) 7 (c) 9 (d) 10
21. Suppose `MAX_SIZE` is the size of the array used in the implementation of circular queue, array index starts with 0, `Front` points to the first element in the queue, and `Rear` points to the last element in the queue. Which of the following condition specifies that circular queue is EMPTY?
- (a) $\text{Front} = \text{Rear} = 0$ (b) $\text{Front} = \text{Rear} = -1$
(c) $\text{Front} = \text{Rear} + 1$ (d) $\text{Front} = (\text{Rear} + 1) \% \text{MAX_SIZE}$
22. A data structure in which elements can be inserted or deleted at/from both the ends but not in the middle is:
- (a) Queue (b) Circular queue (c) Deque (d) Priority queue
23. A normal queue, if implemented using an array of size `MAX_SIZE`, gets full when:
- (a) $\text{Rear} = \text{MAX_SIZE} - 1$ (b) $\text{Front} = (\text{rear} + 1) \bmod \text{MAX_SIZE}$
(c) $\text{Front} = \text{rear} + 1$ (d) $\text{Rear} = \text{front}$
24. An array of size `MAX_SIZE` is used to implement a circular queue. `Front`, `Rear`, and `Count` are tracked. Suppose `Front` is 0 and `Rear` is `MAX_SIZE - 1`. How many elements are present in the queue?
- (a) Zero (b) One (c) $\text{MAX_SIZE} - 1$ (d) `MAX_SIZE`
25. Which one of the following is an application of queue data structure?
- (a) When a resource is shared among multiple consumers.
(b) When data is transferred asynchronously (data not necessarily received at the same rate on sending) between two processes.
(c) Load Balancing (d) All of these
26. How many queues are needed to implement a stack? Consider the situation where no other data structure like an array, a linked list is available to you.
- (a) 1 (b) 2 (c) 3 (d) 4

27. How many stacks are needed to implement a queue? Consider the situation where no other data structure like an array, a linked list is available to you.
- (a) 1 (b) 2 (c) 3 (d) 4

Answers of MCQs

01 (a)	02 (d)	03 (a)	04 (a)	05 (a)	06 (b)	07 (c)	08 (d)	09 (d)	10 (b)
11 (c)	12 (d)	13 (a)	14 (b)	15 (a)	16 (a)	17 (a)	18 (c)	19 (b)	20 (a)
21 (b)	22 (c)	23 (a)	24 (d)	25 (d)	26 (b)	27 (b)			

- Q.2 Differentiate between the following:
- i) Queue & Deque ii) Priority Queue & Queue
- Q.3 Write an algorithm and program to find the number of items in a queue.
- Q.4 Write an algorithm to reverse the data items stored in a queue.
- Q.5 What is a deque? Write an algorithm to insert items into a deque.
- Q.6 Write an algorithm to remove and insert items in a deque.
- Q.7 What is a circular queue? Explain with examples.
- Q.8 What is the priority queue? Explain the implementation of the priority queue in programming languages.
- Q.9 Write a program to insert five integer values into a queue. Remove the values from the queue one by one and display only the even values on the screen.
- Q.10 Write a program to insert 15 values into a queue. Remove the values from the queue one by one. Find the maximum & minimum values and display results on the screen.
- Q.11 Write a program to insert five integer values into a deque. Remove all the values from the deque. Calculate their average and display the result on the screen.
- Q.12 Write an algorithm and program to insert ten patient names into a priority queue. Display all the patient names from the priority queue that have the top priority.

Searching and Sorting are the fundamental requirements in all types of computer applications. These are very important operations in computer science and are mostly performed on data structures like arrays, linked lists, and trees. This chapter presents techniques for implementing searching and sorting using array data structures.

8.1 SEARCHING

Computer systems are often used to store large amounts of data from which specific data items or individual records are needed to be retrieved according to some search criteria. Organizing and retrieving information are very important tasks in most computer applications. Searching is the most frequently performed operation in all types of applications or computing tasks. The process of finding a specific data item or record from a given list is called *searching*.

The search is successful if the specified data item is found during the searching process and is declared unsuccessful otherwise. The search operation is terminated as soon as the required data item is found; however, it is possible that more than one instances of the search item exist in the given list.

Efficient storage and searching of data are very important from the programmers' point of view. Various searching algorithms have been introduced by the professionals but only efficient ones are adopted in real-world applications. Different operations on data like insertion, deletion, updating data, etc. are normally performed after the search operation. For example, to delete a data item from a list, it is first searched (or located) in the list and then the deletion operation is performed.

The records in a database are searched based on their key field values. The database records can be searched randomly as well as sequentially.

Search Methods

A variety of search methods can be used (depending on the situations) for searching information. The most commonly used search methods are as follows:

- i. Sequential Search
- ii. Binary Search

8.1.1 Sequential Search

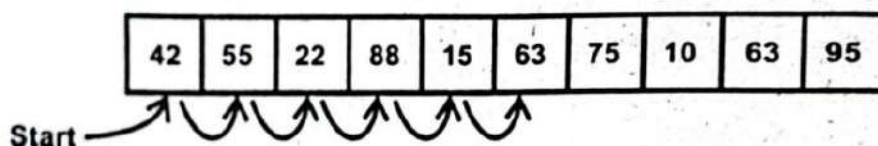
A sequential search is also known as *serial search* or *linear search*. It is a very simple and straight forward technique to search a specific data item in an unordered list. The specific data item is searched in the list sequentially one by one in a sequence, i.e. starting from the first data item up to the required data item of the list or end of the list is reached.

Using a sequential search, the following procedure is adopted:-

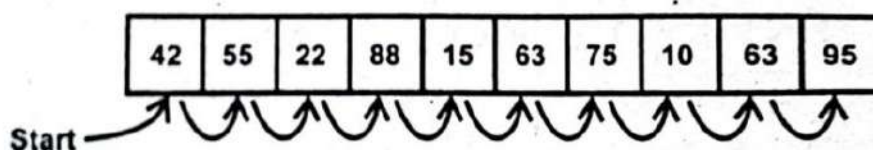
- ★ The required value is compared with the first value of the list.
If the required value matches with the first value of the list, the search operation is declared successful and is stopped.
- ★ If the required value does not match with the first value of the list, it is compared with the second value of the list. The search process continues until the value is found or the end of the list is reached.
- ★ If the end of the list is reached without finding the required value, the search process terminates unsuccessfully.

Example:

An array of 10 elements filled with data values is shown in the figure. Suppose we want to search the value 63 in this array. We can see this value exists at locations 6 and 9. Therefore, the search operation will be terminated at position 6.



Now, suppose we want to search another value 66 in this array. This value does not exist in this array. The search operation will be terminated at the end of the array.



Comparisons in Sequential Search

When the required value is present in the first position of the array, the number of comparisons will be 1. When the required value is at position K of the array, the number of comparisons will be K. Similarly, when the required value is in the last position of the array, the number of comparisons will be N, where N is the size of the array. Thus, a minimum number of comparisons in the sequential array will be 1 and the maximum number of comparisons will be N. Hence, the average number of comparisons in a sequential search will be $(N+1)/2$.

Please note that the performance of linear search improves if the required value is near to the beginning of the array (list) than to its end because a fewer number of comparisons are needed in this situation.

Application of Sequential Search

The sequential search, however, is a slow process. Because, in the case of large lists, a large number of comparisons are needed to search the required value. Therefore, this search method is used for only small lists of data. The sequential search method is mostly used in:

- Application programs for finding specific text or word(s) in the documents such as finding text in MS-Word or information downloaded in the web browser window.
- Finding records or information stored in a sequential file.

The sequential search method is not recommended for a large amount of data because some more efficient search methods are available for large and complex searches. These search methods are *binary search* and *hash table*.

Algorithm – Sequential Search

Write an algorithm that searches a value from an array 'XYZ' having N elements.

1. START
2. SET LOC = -1
3. INPUT N values into an array XYZ
INPUT VAL [Enter value that is to be searched]
4. REPEAT Step-5 FOR I = 1 TO N
5. IF VAL = XYZ[I] THEN
LOC = I
PRINT "Value found at location", LOC
EXIT
END IF
[End of step-4 Loop]
6. IF LOC = -1 THEN
PRINT "Value not found"
END IF
7. EXIT

Program Code 8.1

```
// Program to search a value in an array using sequential search
#include <iostream.h>
#include <conio.h>

class seq_search
{
    private:
        int xyz[5];
    public:
        void input(void);
        void search(int);
};

void main(void)
{
    seq_search obj;
```



```

    int val;
    clrscr();
    obj.input();
    cout<<"Enter a value to search : ";
    cin>>val;
    obj.search(val);
    getch();
} //end of main()

//member function to input data into array
void seq_search::input(void)
{
    cout<<"Enter 5 values"<<endl;
    for(int i = 0; i<=4; i++)
        cin>>xyz[i];
} //end of input()

//member function to search data from array
void seq_search::search(int n)
{
    int i, loc = -1;
    i = 0;
    while(i<=4)
    {
        if(n == xyz[i])
        {
            loc = i + 1;
            cout<<"Value found at location = "<<loc;
            break;
        }
        i++;
    } //end of while loop
    if(loc == -1)
        cout<<"Value not found";
} //end of search()

```

Output of the program:

```

Enter 5 values
25
6
8
9
75
Enter required value to search : 8
Value found at location = 3

```

Algorithm - Searching & Replacing using Sequential Search

Write an algorithm that searches a value in an array XYZ having N elements and replaces the searched value with a new one. Use a sequential method to search the array.

1. START
2. SET LOC = -1
3. INPUT N values into array XYZ
4. INPUT VAL1 [Enter a value to be searched into variable VAL1]
INPUT VAL2 [Enter a value to be replaced with VAL1 into variable VAL2]
5. REPEAT Step-6 FOR I = 1 TO N
6. IF VAL1 = XYZ[I] THEN
 XYZ[I] = VAL2
 LOC = I
 PRINT "Value is modified at location", LOC
 EXIT
END IF
[End of Step-5 loop]
7. IF LOC = -1 THEN
 PRINT "Value not Found"
 EXIT
END IF
8. EXIT

Program Code 8.2

// Program to search a value in an array and replace it with another one.

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
class seq_search
{
private:
    int xyz[5];
public:
    void input(void);
    void search_replace(int, int);
};
```

```
void main(void)
{
    seq_search obj;
    int val1, val2;
    clrscr();
    obj.input();
    cout<<"Enter required value to search : ";
```



```

        cin>>val1;
        cout<<"Enter value to replace : ";
        cin>>val2;
        obj.search_replace(val1, val2);
        getch();
    } //end of main()

    //member function to input data into array
    void seq_search::input(void)
    {
        cout<<"Enter 5 values "<<endl;
        for(int i = 0; i<=4; i++)
            cin>>xyz[i];
    } //end of input()

    //member function to search value from array and replace it with new one
    void seq_search::search_replace(int m, int n)
    {
        int i, loc = -1;
        i=0;
        while(i<=4)
        {
            if(m == xyz[i])
            {
                xyz[i] = n;
                loc = i + 1;
                cout<<"Value is modified at location = "<<loc;
                break;
            }
            i++;
        } //end of while loop
        if(loc == -1)
            cout<<"Value not found";
    } //end of search_replace()

```

Output of the program:

```

Enter 5 values
15
24
36
85
74
Enter required value to search : 85
Enter value to replace : 97
Value is modified at location = 4

```

Algorithm – Searching the largest value

Write an algorithm that finds the largest value in an array 'XYZ' consisting of 5 elements.

1. START
2. INPUT five values into array XYZ
3. [Assigns value of first element of array to MAX, and 1 to C]
MAX = XYZ[0], C = 1
4. [Loop that searches maximum value in an array]
REPEAT Step-5 WHILE C ≤ 4
5. IF MAX < XYZ[C] THEN
 MAX = XYZ[C]
 C = C + 1
END IF
[End of loop of step- 4]
6. PRINT MAX
7. EXIT

Program Code 8.3

// Program to find the largest value in an array using sequential search method

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
class max_value
{
    private:
        int xyz[5];
    public:
        void input(void);
        int search(int&);
};
```

```
void main(void)
```

```
{
    max_value obj;
    int pos, max;
    clrscr();
    obj.input();
    max = obj.search(pos);
    cout<<"The largest value is: "<<max
        <<" which is present at location = "<<pos;
    getch();
} // end of main()
```

```
//member function to input value into array
```



```

void max_value::input(void)
{
    cout<<"Enter 5 values "<<endl;
    for(int i = 0; i<=4; i++)
        cin>>xyz[i];
} // end of input()

//member function to find the largest value and its location in an array
int max_value::search(int &loc)
{
    int m, i = 0;
    m = xyz[0];
    while(i<=4)
    {
        if(m < xyz[i])
        {
            m = xyz[i];
            loc = i+1;
        }
        i++;
    } // end of while loop
    return m;
} // end of search()

```

Output of the program:

Enter 5 values

25

41

21

3

21

The largest value is: 41, which is present at location = 2

8.1.1.1 Complexity Analysis of Sequential Search Algorithm

In a sequential search, when the required value is in the last position of the array or does not exist in the array, then the number of comparisons will be 'n', where 'n' is the size of the array. Therefore, the worst-case time complexity will be $O(n)$. The average number of comparisons will be $\frac{n+1}{2}$ (i.e. $\frac{1+2+3+\dots+n}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2} \approx n$). Therefore, the time complexity and space complexity of sequential search algorithm are as follows:

Time Complexity		Space Complexity
Average-Case	Worst-Case	Worst-Case
$\Theta(\frac{n+1}{2}) \approx \Theta(n)$	$O(n)$	$O(n)$

1.2 Binary Search

The binary search method is very fast as compared to a sequential search. This search method is applied on the sorted array or list to search a specific data value. This search method is used to search a large-size list in finding a specific value.

The binary search method works on the principle of "divide and conquer". The given list is divided into two equal halves. The search process is started in the middle of the sorted list. If the required value is in the middle of the list then the search process is successful and is stopped at that point; otherwise, one of the two halves of the list is selected for further search. The main idea of the binary search is to repeatedly cut a list into two halves in every comparison.

Example:

Consider an array "ABC" with 10 values in sorted order, as shown in the following figure:

Start					Mid					End
↓					↓					↓
2	4	7	12	23	38	44	65	87	99	
0	1	2	3	4	5	6	7	8	9	

Following steps explain the binary search technique for finding a specific value in the above array. Suppose we want to search the value 12 in this array.

- Step-1:** Divide the array into two halves such as " $\text{Mid} = (\text{Start} + \text{End})/2$ ". In the above case, the value of Start is 0, while End is 9. So the value of Mid is 4. The value $\text{Mid}[4]$, i.e. 23 is greater than 12. The required value i.e. 12 exists on the left half side of the array i.e. 0 to 3.
- Step-2:** Divide again the left side of the array from 0 to 3. In this case, new values of Start and End are 0 and 3 ($\text{End} = \text{Mid} - 1$) respectively. $\text{Mid} = (\text{Start} + \text{End})/2$. So the new value of Mid is 1. The value $\text{Mid}[1]$ i.e. 4 is less than 12. The required value i.e. 12 exists on the right half side i.e. 2 to 3.
- Step-3:** Divide again array from 2 to 3. In this case, new values of Start and End are 2 and 3 ($\text{Start} = \text{Mid} + 1$) respectively. $\text{Mid} = (\text{Start} + \text{End})/2$. So the new value of Mid is 2. The value $\text{Mid}[2]$ i.e. 7 is less than 12. The required value i.e. 12 exists on the right half side i.e. 3 to 3.
- Step-4:** Divide again array from 3 to 3. In this case, new values of Start and End are 3 and 3 ($\text{Start} = \text{Mid} + 1$) respectively. $\text{Mid} = (\text{Start} + \text{End})/2$. So the new value of Mid is 3. The value $\text{Mid}[3]$ i.e. 12 is equal to 12. The value is located at position 3. The search process is terminated.

	0	1	2	3	4	5	6	7	8	9
Step-1	2	4	7	12	23	38	44	65	87	99
Step-2	2	4	7	12						
Step-3			7	12						
Step-4				12						

Examples of Binary Search in Real World

The binary search method has many applications in real-world situations. Some of these applications are as follows:

- **Searching a specific page in a book:** Suppose we are searching for page 6 in a book of 20 pages. First, we open the book from the middle. If the current page is less than 6, we open at a page to the right; otherwise, we open at a page to the left. We repeat this process until page 6 is found. In this example, we have a sorted list of size 20. The first comparison is with the middle page number 10. This eliminates the last 10 pages, as page 6 is less than 10. The second comparison is from page 1 to page 10. This eliminates page 1 to page 5, as page 6 is greater than page 5. This process continues until page 6 is found.
- **Searching a specific word in the dictionary:** For searching a particular word in the dictionary, we usually start from the middle in the dictionary. If the word that we are searching for comes before words on the current page, it shows that the word should be before this page. So we look at the first half. Otherwise, we search for the word in the second half of the dictionary. Suppose the word is in the first half of the dictionary, we consider the first half for looking at the word. We do not need to look into the second half of the dictionary. Thus the data to be searched is divided into half of the process. Now we divide this portion into two halves and resume search for the word. Here we again have to find whether the word is in the first half or the second half of this portion. The same step is repeated with the part that contains the required word. Finally, we come to the page where the required word exists. We see that in the binary search, the search process speeds up because of dividing the target list into two halves repeatedly.

Algorithm – Binary Search

Write an algorithm that finds a value in an array 'ABC' consisting of 10 elements, sorted in ascending order. Assume that S represents the first element of the array and E represents the last element of the array.

1. START
2. S = 1, E = 10 [Assign values to variables S & E]
3. SET LOC = - 1
4. INPUT values in array ABC in ascending order.

5. INPUT VAL [Enter value to be searched in variable VAL]
6. [Loop that searches value in an array using Binary search]
- REPEAT Step-7 and Step-8 WHILE S <= E
7. MID = (S + E)/2
8. IF VAL = ABC[MID] THEN
 - LOC = MID + 1
 - EXIT
 ELSE IF VAL < ABC[MID]
 - E = MID - 1
 ELSE
 - S = MID + 1
 END IF
 [End of Step-6 loop]
9. IF LOC = -1 THEN
 - PRINT "Value not found:"
 ELSE
 - PRINT "Value found at location = ", LOC
 END IF
10. EXIT

Program Code 8.4

//Program to search a required value from array with binary search method

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
class bin_search
{
    private:
        int abc[10];
    public:
        void input(void);
        void search(int);
};
```

```
void main(void)
```

```
{
    bin_search obj;
    int val;
    clrscr();
    obj.input();
    cout<<"Enter value to search : ";
    cin>>val;
    obj.search(val);
    getch();
} // end of main()
```



```

//member function to enter values into array
void bin_search::input(void)
{
    cout<<"Enter 10 values into array in ascending order: \n";
    for(int i = 0; i<=9; i++)
        cin>>abc[i];
} // end of input()

//member function to search a value from array
void bin_search::search(int num)
{
    int loc, M, S = 0, E = 9;
    loc = -1;
    while(S<=E)
    {
        M = (S + E)/2;
        if(num == abc[M])
        {
            loc = M + 1;
            break;
        }
        else if(num < abc[M])
            E = M - 1;
        else
            S = M + 1;
    } // end of while loop
    if(loc == -1)
        cout<<"Value not found"<<endl;
    else
        cout<<"Value found at position = "<<loc<<endl;
} // end of search()

```

Output of the program:

Enter 10 values into array in ascending order:

1
3
9
15
78
87
95
103
124
352

Enter value to search : 87

Value found at position = 6

Complexity Analysis of Binary Search Algorithm

The binary search algorithm divides the working area in half with each iteration. This algorithm runs in at the worst logarithmic time, making $O(\log n)$ comparisons, where 'n' is the number of elements in the array.

The complexities $O(1)$ and $O(n)$ are simple and straightforward. $O(1)$ means an operation is done to reach an element directly (like accessing a specific element of an array), while $O(n)$ means first we would have to search it by checking 'n' elements. However, the complexity $O(\log n)$ is different than $O(1)$ and $O(n)$. To understand the concept of complexity $O(\log n)$, consider the following sorted array of size 16.

0	3	5	8	12	13	15	16	18	20	22	30	40	50	55	66
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Suppose we want to search the number 13 in the above array. We divide the array into halves i.e. sub-arrays and select the middle element as pivot. Since 13 is less than the pivot, select the left half side of the array.

0	3	5	8	12	13	15	16
0	1	2	3	4	5	6	7

$$16 \times \frac{1}{2} = 8$$

We repeat the process of finding the middle element of every sub-array.

12	13	15	16
4	5	6	7

$$8 \times \frac{1}{2} = 4$$

12	13
4	5

$$4 \times \frac{1}{2} = 2$$

13
5

$$2 \times \frac{1}{2} = 1$$

So, for searching a required value from an array of size 16, we have to divide this array 4 times. Mathematically, we can write as:

$$16 \times \left(\frac{1}{2}\right)^4 = 1$$

Similarly, for searching a required value from an array of size 32, we have to divide this array 5 times, an array of size 64, we have to divide this array 6 times, and so on. For searching a required value from an array of size 'n', we have to divide this array 'k' times. The general formula is written as:

$$n \times \left(\frac{1}{2}\right)^k = 1$$

OR

$$n \times \frac{1}{2^k} = 1$$

The above formula can also be written as:

$$2^k \times \frac{n}{2^k} = 2^k \quad (\text{multiplying both sides by } 2^k)$$

$$n = 2^k$$

Now, let us look at the definition of the logarithm, it says that:

A quantity representing the power to which a fixed number (the base) must be raised, produce a given number. So the equation $n = 2^k$ can be written in a logarithmic form as:

$$\log_2 n = k$$

The following table shows the base-2 logarithm with various values of 'N':

N	LOG ₂ N
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10
1,048,576	20
2,097,152	21

The logarithm function grows very slowly. Logarithms are the inverse of exponentials, which grow very rapidly. It makes it easy to calculate the runtime of a binary search algorithm on a 'n' size of the array that's exactly a power of 2. If 'n' is 128, the binary search algorithm divides the array 7 times into sub-arrays.

The time complexity and space complexity of binary search are as follows:

Time Complexity		Space Complexity
Average-Case	Worst-Case	Worst-Case
$\Theta(\log n)$	$O(\log n)$	$O(n)$

6.1.3 Hashing

Hashing is a common technique for storing and searching data in such a way that the data can be inserted and retrieved very quickly. Hashing uses a data structure called a *hash table*. It means that *hashing* is achieved using a *hash table*.

The hashing technique is used for performing insertion, deletion, and search operations at a constant average time. The constant average time means, the amount of time to perform the operation does not depend on data size.

6.1.3.1 Hash Table

A *hash table* is a data structure that stores data items in an associative manner (each data item is stored as "index or key, value" pair). A *hash table* is also known as the *hasp map*. In a hash table, data is stored in an array format, where each data value has its own unique index value. Each position of the *hash table* is often called a *slot* or *bucket*, where the data item is stored (i.e. *hash table* is an array of slots or buckets). Access of data becomes very fast if we know the index or key of the desired data. In C++, usually we implement a *hash table* as an array of linked lists.

A *hash table* has two parts: an array as a storage medium (the actual table where data is stored) and a mapping function, known as a *hash function* which is used for mapping between data items and slots.

6.1.3.2 Hash Function

A *hash function* decides where to store and retrieve items in a *hash table*. It takes a data item (key-value) as its parameter and generates an index location (hash value) for that particular item to store it into the table. Similarly, the *hash function* regenerates (re-computes) the index (hash value) for a particular data item to retrieve it from the table.

There are many possible ways to construct a *hash function*. Different hash functions use different techniques to generate the indices or hash values. Generally, a *hash function* uses modulo arithmetic operator (%) to generate or regenerate the index. In particular, the value of a data item is divided (using modulo % operator) by the table length to generate an index number or hash value in the table. This index number refers to a location, or bucket in the *hash table*, where to store or retrieve the data item.

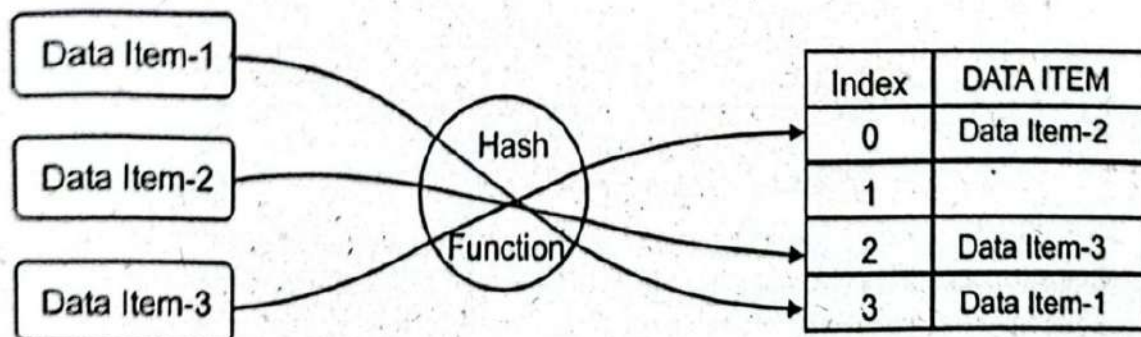


Figure: Working of Hash Function

Example-1: (Hash Table with Integers as Data Items)

Assume that we have a set of integer data items 54, 26, 93, 17, 77, and 31. The *hash function* takes an item and divides it by the table or array size, returning the remainder as its *hash value*. Suppose the size of the array is 11. So the formula to generate an *index* or *hash value* by the simple *hash function* is as under:

$$\text{Index} = (\text{Data Item}) \% 11$$

Therefore, the hash function to compute or generate the index or hash value for integer data item is as under:

```
int hash(int item, int array_size)
{
    return item % array_size;
}
```

Following table gives all of the hash values (indices) of the above data items:

Data Item	Data Item % 11	Hash Value
54	54 % 11	10
26	26 % 11	4
93	93 % 11	5
17	17 % 11	6
77	77 % 11	0
31	31 % 11	9

Inserting Data Items into Hash Table

Once the hash values have been generated (or computed), we can insert each item into *hash table* at the designated position as shown in the figure below. Note that 6 of the 11 slot now occupied. This is referred to as the *load factor*.

0	1	2	3	4	5	6	7	8	9	10
77	(None)	(None)	(None)	26	93	17	(None)	(None)	31	54

Searching Data Items from Hash Table

Now, when we want to search for a data item, we simply use the *hash function* to compute the index or hash value for the data item to be searched. On the basis of the *hash value* we directly go to the slot where this data item is stored. Suppose, in the above example, we want to search 26 from the hash table. We will re-compute its index or hash value and that will be (26%11). Thus, we shall go to slot number 4 and directly access the data item stored in this

We can observe that this searching operation is very fast because a constant amount of time is used to retrieve a data item from any location of a *hash table*.

Example-2: (Hash Table with Strings as Data Items)

Let's take another example of a *hash table* with strings as data items. Suppose the size of the *hash table* is 6, as shown in the following figure:

Hash Table of Strings	
0	(Null)
1	(Null)
2	(Null)
3	(Null)
4	(Null)
5	(Null)

Figure: The empty hash table of strings

In this case, we construct another *hash function* for storing and retrieving strings in the *hash table*. This function computes the sum of the ASCII characters that make up the string and then divides the sum (using % operator) with the size of the table. Therefore, the returned *hash value* will be "sum % table_size".

Suppose we want to store "PAKISTAN" in this table. Following table represents ASCII values of characters that make up the string "PAKISTAN":

Character	ASCII Value
P	80
A	65
K	75
I	73
S	83
T	84
A	65
N	78

The sum of ASCII values of characters of the string "PAKISTAN" is:

$$80 + 65 + 75 + 73 + 83 + 84 + 65 + 78 = 603$$

The hash value for PAKISTAN is $603 \% 6 = 3$ (size of table is 6). Therefore, we insert "PAKISTAN" at index 3.

Hash Table of Strings	
0	(Null)
1	(Null)
2	(Null)
3	PAKISTAN
4	(Null)
5	(Null)

Figure: The hash table after inserting "PAKISTAN"

Let's try another string "CHINA". Following table represents ASCII values of characters that make up the string "CHINA":

Character	ASCII Value
C	67
H	72
I	73
N	78
A	65

The sum of ASCII values of characters of the string "CHINA" is:

$$67 + 72 + 73 + 78 + 65 = 355$$

The hash value for CHINA is $355 \% 6 = 1$. Therefore, we insert "CHINA" at index 1.

Hash Table of Strings	
0	(Null)
1	CHINA
2	(Null)
3	PAKISTAN
4	(Null)
5	(Null)

Figure: The hash table after inserting "CHINA"

The hash function to compute or generate the index or hash value for the string data is as under:

```
int hash(char *str, int table_size)
{
```

```

int sum;

// Make sure a valid string is passed in
if (str==NULL)
    return -1;

// Sum up all the characters in the string
for(; *str; str++)
    sum+= *str;

// Return the sum mod the table size
return sum % table_size;
}

```

8.1.3.3 Collisions in Hashing

A *hash function* does not guarantee that every input will map to a different output. Practically, there is always a chance that two inputs will hash to the same output. This indicates that both data items should be inserted at the same place in the array, but this is impossible. A situation when two or more data items map to the same location in the *hash table* is known as a *collision*. The collision creates a serious problem for the hashing technique. Collisions can be reduced by designing a good hash function.

For example, if a table size is 9, then both the numbers 11 and 20 hash to index 2. Similarly, if a table size is 13, then both the strings "tar" and "rat" will hash to index 2.

8.1.3.4 Collisions Resolution

When two items hash to the same slot, we must have a systematic method for placing the second item in the *hash table*. This process is called *collision resolution*. If the hash function is perfect, collisions will never occur but it is not always possible to design a perfect hash function that would generate a unique index.

There are many collision resolution strategies to handle collisions. Some common strategies are as follows:

- i) Open hashing or separate chaining
- ii) Closed hashing or open addressing

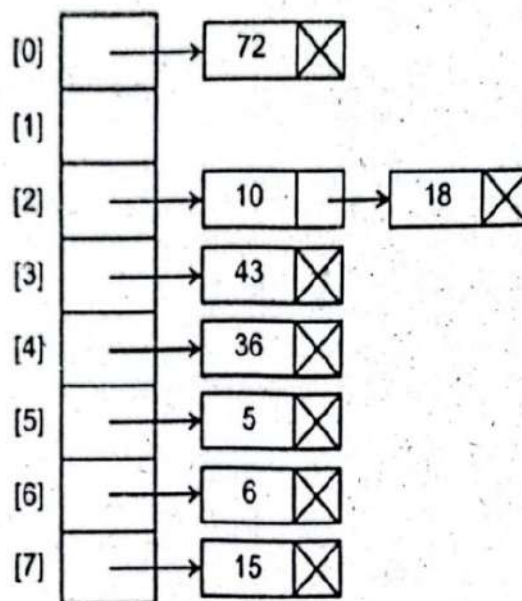
i) Open Hashing (Separate Chaining)

The open hashing is also known as *separate chaining*. In this technique, data items are not directly stored in the slots of the *hash table*. Data items are stored in linked lists. The linked lists are attached to the slots of the *hash table*. The starting address of the linked list is stored in the related slot of the *hash table*. It means that slots of the hash table will be the pointers to linked lists created for the data items. Each slot of *hash table* points to one of these linked lists. When an element hashes to a value, it is added to the linked list at that index in the array. If more than one element hashes to the same value, then these data items are stored in the same linked list. In this

technique, when a specific data needs to be searched, it might become necessary to traverse all the nodes in a specific linked list attached with a hash table to retrieve the data.

Example:

The open hashing technique is shown in the following figure. Suppose the data values that are to be stored in the *hash table* are 10, 15, 18, 72, 5, 6, 43, and 36. The *hash table* is an array with 8 elements. Each element is a pointer to a linked list of numeric data. The *hash function* computes the index of the slot of *hash table* to insert or retrieve data items to and from the linked list attached to the slot. For example, to insert 43, we divide 43 by 8 ($43 \% 8 = 3$). The result is 3. Thus, 43 goes on the list starting at index 3 of the *hash table*. Similarly, we search and delete a specific data item in the *hash table*.



A chain is simply a linked list of all the elements with the same hash key.



The open hashing technique is most commonly used for resolving collisions.

ii) Closed Hashing or Open Addressing

The closed hashing is also known as *open addressing*. In this technique, data items are stored in the *hash table* itself without the use of linked lists. In this method, a hash collision is resolved by linear probing, or searching through alternate locations in the array (the *probe sequence*) until either the target record is found, or an unused array slot is found (It means that the data item is placed into the closed empty slot of the *hash table*).

Example:

Suppose the data values that are to be stored in the *hash table* are 1, 2, 42, 4, 12, 14, 17, 13, and 37. The *hash table* is an array with 20 elements. Following table shows the closed hashing technique:

Sr #	Data Item	Hash	Array Index	Array Index After Probing
1	1	$1 \% 20 = 1$	1	1
2	2	$2 \% 20 = 2$	2	2
3	42	$42 \% 20 = 2$	2	3
4	4	$4 \% 20 = 4$	4	4
5	12	$12 \% 20 = 12$	12	12
6	14	$14 \% 20 = 14$	14	14
7	17	$17 \% 20 = 17$	17	17
8	13	$13 \% 20 = 13$	13	13
9	37	$37 \% 20 = 17$	17	18

Algorithm – Inserting/searching integer values into/from a hash table

Write an algorithm that inserts N integer data values into a hash table and then searches a specific value from the hash table. Take an array ARR of size N.

1. START
2. Initialize ARR array to NULL
3. INPUT N values into array ARR
4. REPEAT Step-5 to Step-7 FOR I = 1 TO N
5. INPUT data value in VAL
6. [Compute the index to store the value into hash table]
INDEX = VAL / N
7. [If the calculated index has empty space, store the value, otherwise display "Collision" message]
IF ARR[INDEX] = NULL THEN
 ARR[INDEX] = VAL
ELSE
 PRINT "Collision Occurs"
END IF
[End of Step-4 loop]
8. INPUT data value to search in X
9. [Search the value from hash table by re-computing its index in hash table]
INDEX = X/N
10. IF ARR[INDEX] = NULL THEN
 PRINT "Value not found"
ELSE
 PRINT "Data value is" ;ARR[INDEX]
END IF
11. EXIT

Program Code 8.5

```
// program to input 10 integer values into a hash table and search a specific value from
// the hash table
#include <iostream.h>
```



```

#include <conio.h>

class hash_integers
{
    private:
        int index;
        int hash_table[10];
    public:
        // Prototypes or declarations of the functions in the class
        void assign_null(void);
        void input(int);
        void search(int);
};

void main(void)
{
    hash_integers hash_obj;
    int val, x;
    clrscr();
    hash_obj.assign_null();
    for(int i = 0; i <= 9; i++)
    {
        cout << "Enter value in slot "<< i << " of hash table: ";
        cin >> val;
        hash_obj.input(val);
    }
    cout << "\n\n Enter a value to search: ";
    cin >> x;
    hash_obj.search(x);
    getch();
} // end of main()

// member function to assign null to all elements of array (hash table)
void hash_integers::assign_null()
{
    for(int i = 0; i <= 9; i++)
        hash_table[i] = NULL;
} // end of assign_null()

// member function to input a value into the hash table
void hash_integers::input(int n)
{
    index = n % 10;
    if(hash_table[index] == NULL)
        hash_table[index] = n;
    else
    {
        cout << "Collision occurs, this value is not stored "<< endl;
        getch();
    }
} // end of input()

```

```

// member function to search a value from the hash table
void hash_integers::search(int n)
{
    index = n % 10;
    if(hash_table[index] != n || hash_table[index] == NULL)
        cout<<"Value not found";
    else
        cout<<"Value found:" <<hash_table[index];
} // end of search()

```

Algorithm – Inserting/searching strings into/from a hash table

Write an algorithm that inserts N strings into a hash table and then searches a specific string from the hash table. Take an array ARR of size N.

1. START
2. Initialize STRINGS array to NULL
3. INPUT N strings into STRINGS array
4. REPEAT Step-5 to Step-7 FOR I = 1 TO N
5. INPUT string in STR
6. [Compute the index to store a string into the hash table by summing up the ASCII values of characters of the string. For this purpose, call function SUM_ASCII and divide the return value of this function by N i.e. size of table]
INDEX = SUM_ASCII(STR)/ N
7. [If the calculated index has empty space, store the string; otherwise display "Collision" message]
IF STRINGS[INDEX] = NULL THEN
 STRINGS[INDEX] = STR
ELSE
 PRINT "Collision Occurs"
END IF
[End of upper loop of Step-4]
8. INPUT string to search in X
9. [Search the string from hash table by re-computing its index in hash table]
INDEX = SUM_ASCII(X)/ N
10. IF STRINGS[INDEX] = NULL THEN
 PRINT "Value not found"
ELSE
 PRINT "Data value is" ; STRINGS[INDEX]
END IF
11. EXIT

Function SUM_ASCII(STR1)

1. C = 0, SUM = 0
2. REPEAT Step-3 to Step-4 WHILE STR1[C] != NULL


```

3.      SUM = SUM + ASCII value of STR1[C]
4.      C = C + 1
        [End of while loop]
5.      RETURN SUM
6.      EXIT

```

Program Code 8.6

// program to input 5 strings into a hash table and search a specific string from the hash table

```

#include <iostream.h>
#include <conio.h>
#include <string.h>

class hash_strings
{
    private:
        int index;
        char hash_table[5][25];    // string array of 5 elements
    public:
        // Prototypes or declarations of the functions in the class
        void assign_null(void);
        int sum_ascii(char *);
        void input(char *);
        void search(char *);
};

void main(void)
{
    hash_strings hash_obj;
    char str[24];
    clrscr();

    // assign null to all elements of string array
    hash_obj.assign_null();

    // input 5 strings into string array
    for(int i = 0; i <= 4; i++)
    {
        cout << "Enter string in slot "<< i << " of hash table: ";
        cin >> str;
        hash_obj.input(str);
    }

    cout << "\n\n Enter a string to search: ";
    cin >> str;
    hash_obj.search(str);
    getch();
} // end of main()

// member function to assign null value to all elements of string array (hash table)
void hash_strings::assign_null()
{
    for(int i = 0; i <= 4; i++)
        strcpy(hash_table[i], NULL);
} // end of assign_null()

```

```

// member function to assign a string to a string array (hash table)
void hash_strings::input(char * str1)
{
    index = sum_ascii(str1) % 5;
    if(strcmp(hash_table[index], NULL)==0)
        strcpy(hash_table[index],str1);
    else
        cout<<"Collision occurs, this string is not stored "<<endl;
} //end of input()

// member function to search a value from the hash table
void hash_strings::search(char * str1)
{
    index = sum_ascii(str1)% 5;
    if(strcmp(hash_table[index], str1)!= 0 ||
        strcmp(hash_table[index], NULL)==0)
        cout<<"String not found";
    else
        cout<<" String found:"<<hash_table[index];
} //end of search()

// member function to sum up the ASCII values of the characters of the string
int hash_strings::sum_ascii(char * str1)
{
    int c = 0, sum = 0;
    for(c = 0; str1[c]!='\0'; c++)
        sum = sum + str1[c];
    return sum;
} //end of sum_ascii()

```

1.1.3.5 Complexity Analysis of Hash Table

The time complexity and space complexity of various operations on the hash table are as follows:

Operation	Time Complexity		Space Complexity
	Average-Case	Worst-Case	Worst-Case
Search	$O(1)$	$O(n)$	$O(n)$
Insertion	$O(1)$	$O(n)$	$O(n)$
Deletion	$O(1)$	$O(n)$	$O(n)$

8.2 SORTING

The process of arranging data items in a list in a specific order is called *sorting*. The order of data items in the list may be in ascending or descending order. Ascending order means that the values are arranged from smaller to larger values. For example, a list of values 25, 6, 30, 42, and 3 in ascending order is:

3, 6, 25, 30, 42

In descending order, values are arranged in reverse order, i.e. larger to smaller values. For example, a list of values 25, 6, 30, 42, and 3 in descending order is:

42, 30, 25, 6, 3

Similarly, alphabets in descending order may be written as;

Z, Y, X, W,, C, B, A

Sorting is one of the most important operations performed in computer systems. It is mostly used to arrange records in databases. A record in a database consists of several fields. It contains one (or more) field whose value uniquely determines a record in the database file. Such a field is called the *key field* or the *primary key*. The value that a key field holds is called the *key value*. Sorting operation is normally performed on the key values of records.

For example, in a database consisting of records of students, the key field may be "roll numbers" of students. The value in this field is always unique as no two students can have the same roll number in the same class. The records of the students' database can be sorted according to the class roll numbers. However, some other fields like "obtained marks" may also be used for sorting records.

Sorting Efficiency

There are many techniques for sorting. The implementation of a particular sorting technique depends upon the situation. Sorting techniques mainly depend on two parameters. The first parameter is the execution time of the program, which means time taken for execution of the program. Second is the space, which means space taken by the program in storage device i.e. memory.

Sorting Methods

There are different methods for sorting the data. However, we shall discuss here the most important and commonly used methods which are as follows:

- | | |
|---------------------|--------------------|
| i) Bubble Sort | ii) Selection Sort |
| iii) Insertion Sort | iv) Shell Sort |

- | | |
|--------------------|-------------------|
| v) Quick Sort | vi) Merge Sort |
| vii) Counting Sort | viii) Bucket Sort |
| ix) Heap Sort | x) Radix Sort |

8.2.1 Bubble Sort

Bubble sort method is the oldest and the simplest sorting method. It is also known as a *sinking sort* or *exchange sort*. It works by comparing each pair of adjacent (neighboring) elements of the array and swapping (exchanging) them if they are not in order. This process continues until the entire array is sorted.

(This sort method is given the name "bubble sort" because values "float like a bubble" from one end of the array or list to another (just like a water bubble rises to the water surface). Suppose we are sorting an array of numbers in ascending order, higher values bubble out to the right whereas lower values bubble out to the left side of the array.

Bubble sort is generally considered to be the most inefficient sorting method because it is very slow for sorting a large amount of data. However, it is simple and can easily be implemented. Usually, it is used for sorting only a small amount of data.

To sort an array of "n" elements, n-1 passes (or iterations) are required. The following steps explain the sorting of data in an array in ascending order using the bubble sort method:

- ★ In the first pass or iteration, the largest value bubbles out to the last position in the array.
- ★ In the second pass, the second largest value bubbles out to the second last position in the array and so on.
- ★ In the 'n-1' pass, the data gets arranged in ascending order.)

Example:

An array with 5 elements and filled with data values is shown in the figure. Assume that the name of the array is 'A'.

6	12	1	7	3
---	----	---	---	---

¹⁻¹²
To sort this array in ascending order using the bubble sort method, 4 passes (i.e. N-1, where N is 5, so 5-1 = 4) will be required. These are described as follows:

Pass -1:

In the first pass, all the values of the array are compared with adjacent or neighboring values i.e. from the first element to the last element of the array; these are compared in the following order:

- ✓ A[0] is compared with element A[1]; as 6 is not greater than 12, there will be no change in the positions of the values i.e. elements are not interchanged.

6	12	1	7	3
---	----	---	---	---

- ✓ A[1] is compared with element A[2]; as 12 is greater than 1, the values are interchanged. The array is rearranged as:

6	1	12	7	3
---	---	----	---	---

- ✓ A[2] is compared with element A[3]; as 12 is greater than 7, the values are interchanged. The values of the array take the following positions:

6	1	7	12	3
---	---	---	----	---

- ✓ A[3] is compared with element A[4]; as 12 is greater than 3, the values are interchanged. The position of the array becomes:

6	1	7	3	12
---	---	---	---	----

Thus at the end of the first pass or iteration, the largest value moves or bubbles out to the last position in the array.

6	12	1	7	3
---	----	---	---	---

6	1	12	7	3
---	---	----	---	---

6	1	7	12	3
---	---	---	----	---

6	1	7	3	12
---	---	---	---	----

✓ Pass -2:

In the second pass, values are compared from the first element to the second last element of the array. These are compared in the following order:

- ✓ A[0] is compared with element A[1]; as 6 is greater than 1, the values are interchanged. The array takes the following position:

1	6	7	3	12
---	---	---	---	----

- ✓ A[1] is compared with element A[2]; here 6 is not greater than 7, there will be no change in the positions of the values. The array remains the same.

1	6	7	3	12
---	---	---	---	----

- $A[2]$ is compared with element $A[3]$; 7 is greater than 3, the values are interchanged. Values take the following positions in the array:

1	6	3	7	12
---	---	---	---	----

Thus at the end of the second pass, the second largest value bubbles out to the second last position in the array.

1	6	7	3	12
1	6	7	3	12
1	6	3	7	12

Pass -3:

In the third pass, values are compared from the first element to the third last element of the array. These are compared in the following order:

- $A[0]$ is compared with element $A[1]$; 1 is not greater than 6, there will be no change in the array. The array remains the same.

1	6	3	7	12
---	---	---	---	----

- $A[1]$ is compared with element $A[2]$; here 6 is greater than 3, the values are interchanged. The array takes the following position:

1	3	6	7	12
---	---	---	---	----

Thus at the end of the third pass, the third-largest value bubbles out to the third last position in the array.

1	6	3	7	12
1	3	6	7	12

Pass -4:

In the fourth pass, values are compared from the first element to the fourth last element of the array. These are compared in the following order:

- $A[0]$ is compared with element $A[1]$; as 1 is not greater than 3, there will be no change in the array. The position of the array remains the same. Thus the array has already been sorted in ascending order.

1	3	6	7	12
---	---	---	---	----

Algorithm – Bubble Sort

Write an algorithm to sort an array 'ABC' in ascending order using the bubble sort method. The array consists of N elements.

1. START
2. SET $U = N$ [U represents the control variable for controlling upper loop]
3. REPEAT STEP-4 TO 8 WHILE ($U \geq 1$) [Start of upper loop]
4. SET $I = 1$
5. REPEAT STEP-6 TO 7 WHILE ($I \leq U$) [Start of inner loop]
6. IF $ABC[I] > ABC[I+1]$ THEN [Interchange values]
 - TEMP = $ABC[I]$
 - $ABC[I] = ABC[I+1]$
 - $ABC[I+1] = TEMP$
7. END IF
8. $I = I + 1$ [End of Inner loop]
9. $U = U - 1$ [End of Upper loop]
10. EXIT

Program Code 8.7

// Program to sort an array with 5 elements using bubble sort method.

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
class bubble
```

```
{
```

```
private:
```

```
int abc[5];
```

```
public:
```

```
void input(void);
```

```
void sort(void);
```

```
void print(void);
```

```
};
```

```
void main(void)
```

```
{
```

```
    bubble obj;
```

```
    clrscr();
```

```
    obj.input();
```

```
    obj.sort();
```

```
    obj.print();
```

```
    getch();
```

```
} //end of main()
```

```

// Member function to input data into array
void bubble::input(void)
{
    cout<<"Enter 5 values "<<endl;
    for(int i = 0; i<=4; i++)
        cin>>abc[i];
} // end of input()

// Member function to sort array
void bubble::sort(void)
{
    int temp;
    for(int u = 4; u>=1; u--)
        for(int i = 0; i<u; i++)
            if(abc[i] > abc[i+1])
            {
                temp = abc[i];
                abc[i] = abc[i+1];
                abc[i+1] = temp;
            }
} //end of sort()

// Member function to print sorted data
void bubble::print(void)
{
    cout<<"Sorted array"<<endl;
    for(int i = 0; i<=4; i++)
        cout<<abc[i]<<" ";
    cout<<endl;
} //end of print()

```

Output of the program:

Enter 5 values

2

74

1

65

22

Sorted array

1

2

22

65

74

8.2.1.1 Complexity Analysis of Bubble Sort Algorithm

The bubble sort algorithm uses a nested loop. It is straightforward to analyze. The upper loop is used for the number of passes. The inner loop is used for comparisons. Suppose we have an unsorted array of size 'n'. The number of passes required to sort this array will be 'n-1'. The number of comparisons in each pass of the bubble sort algorithm:

S#	Number of Comparisons
Pass:1	n-1
Pass:2	n-2
Pass:3	n-3
-----	----
Pass:n-3	n-(n-3) = 3
Pass:n-2	n-(n-2) = 2
Pass:n-1	n-(n-1) = 1

$$\text{Total comparisons} = (n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1$$

It is the arithmetic series. The formula for this arithmetic series:

$$\begin{aligned}
 (n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1 &= \frac{(n-1)(1+n-1)}{2} \\
 &= \frac{(n^2 - n)}{2} \\
 &= \frac{n^2}{2} - \frac{n}{2}
 \end{aligned}$$

As n^2 has a higher growth rate. Hence, the time complexity of the bubble sort algorithm is $O(n^2)$.

The space complexity of the bubble sort algorithm is $O(1)$ because only a single additional memory space is required i.e. for the 'temp' variable. Following table shows the time complexity and space complexity of bubble sort algorithm:

Time Complexity			Space Complexity
Best-Case	Average-Case	Worst-Case	Worst-Case
$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$

8.2.2 Selection Sort

The selection sort is the simplest sorting method. It first finds the smallest value in the array and exchanges this value with the value in the first position of the array. Then, it finds the second smallest value and exchanges this value with the value in the second position of the array, and so on. This process continues until the entire array is sorted.

The selection sort method is relatively easy to implement. It is also slightly faster than bubble sort because selection sort makes only one exchange for every pass through the list or array. It is, however, inefficient for larger lists.

To sort an array of "n" elements, n-1 passes (or iterations) are required. The following steps explain the sorting of data in an array in ascending order using the selection sort method:

- ★ In the first pass or iteration, the whole array is searched to find the smallest value. The element of the array that has the smallest value is selected. This value is exchanged (interchanged) with the first element of the array in order to place the smallest value in the first position.
- ★ In the second pass, the array is searched from the second element to the last element of the array to find the smallest value (i.e. second smallest value). This value is exchanged with the second element of the array and so on.
- ★ This process is repeated until the entire array is sorted.

Example:

An array with 5 elements and along data values is shown in the following figure. Assume that the name of the array is 'A'.

6	12	1	7	3
---	----	---	---	---

To sort this array in ascending order using the selection sort method, 4 passes will be required. These are described as follows:

Pass -1:

The array is searched starting from the first element to the last element. The element that has the smallest value is selected. The smallest value is 1 at position 3. Thus the value at position 3 is exchanged with the value of position 1, i.e. values of A[0] and A[2] are swapped. The array, before and after exchanging values is shown below:

6	12	1	7	3
---	----	---	---	---

Before Swapping

1	12	6	7	3
---	----	---	---	---

After Swapping

Pass -2:

The array is searched starting from the second element to the last element. The element that has the smallest value is selected. The smallest value is 3 at position 5. Thus the value at position 5 is interchanged with the value of position 2, i.e. values of A[1] and A[4] are swapped. The array, before and after exchanging values is shown below:

1	12	6	7	3
---	----	---	---	---

Before Swapping

1	3	6	7	12
---	---	---	---	----

After Swapping

Pass -3:

The array is searched starting from the third element to the last element. The element that has the smallest value is selected. The smallest value is 6 at position 3. Since 6 is a

smaller value and it is already located at position 3, no need to exchange the values. The array after this pass is shown below;

1	3	6	7	12
---	---	---	---	----

Before Swapping

1	3	6	7	12
---	---	---	---	----

After Swapping

Pass -4:

The array is searched starting from the fourth element to the last element i.e. elements 4th and 5th. The element that has the smallest value is selected. The smallest value is 7 at position 4. Since 7 is a smaller value and it is already located at location 4, no need to exchange the values. The array after this pass is shown below. We get the sorted array after the last pass.

1	3	6	7	12
---	---	---	---	----

Before Swapping

1	3	6	7	12
---	---	---	---	----

After Swapping

Algorithm – Selection Sort

Write an algorithm to sort an array ABC in ascending order using the selection sort method. The array consists of N elements.

1. START
2. SET U = 0 [U represents the control variable used for upper loop]
3. REPEAT STEPS 4 TO 10 WHILE (U < N)
4. MINI = ABC[U]
5. I = U [I represents the control variable used for inner loop]
6. REPEAT STEPS 7 TO 8 WHILE (I ≤ N)
7. IF MINI > ABC[I] THEN
 - MINI = ABC[I]
 - LOC = I
8. END IF
9. I = I + 1 [End of Step-6 loop --- Inner Loop]
10. [Interchange values]
 - TEMP = ABC[LOC]
 - ABC[LOC] = ABC[U]
 - ABC[U] = TEMP
11. U = U + 1 [End of Step-3 loop --- upper Loop]
- 1.1. EXIT

Program Code 8.8

```
// Program to sort array with 5 elements using selection sort method
#include <iostream.h>
```

```
#include <conio.h>

class selection
{
    private:
        int abc[10];
    public:
        void input(void);
        void sort(void);
        void print(void);
};

void main(void)
{
    selection obj;
    clrscr();
    obj.input();
    obj.sort();
    obj.print();
    getch();
} //end of main()

// Member function to input data into array
void selection::input(void)
{
    cout<<"Enter 5 values "<<endl;
    for(int i = 0; i<=4; i++)
        cin>>abc[i];
} //end of input()

// Member function to sort an array
void selection::sort(void)
{
    int mini, temp, loc;
    for(int u = 0; u<4; u++)
    {
        mini = abc[u];
        loc = u;
        for(int i = u; i<=4; i++)
            if(mini>abc[i])
            {
                mini = abc[i];
                loc = i;
            }

        // swap values
        temp = abc[loc];
        abc[loc] = abc[u];
    }
}
```



```

        abc[u] = temp;
    }
} //end of sort()

// Member function to print values of an array
void selection::print(void)
{
    cout<<"Sorted array "<<endl;
    for(int i = 0; i<=4; i++)
        cout<<abc[i]<<" ";
    cout<<endl;
} //end of print()

```

Output of the program:

Enter 5 values

25

4

85

1

35

Sorted array

1

4

25

35

85

8.2.2.1 Complexity Analysis of Selection Sort Algorithm

The selection sort algorithm is easy to analyze than other sorting algorithms. It uses a nested loop. The upper loop is used for the number of passes. The inner loop is used for comparisons. Suppose we have an unsorted array of size 'n'. The number of passes required to sort this array will be 'n-1'. In the first pass, it finds the lowest value from 'n' elements (from 0 to n) of the array and then swaps it with the first element of the array. The first pass takes 'n-1' comparisons. In the second pass, it finds the next lowest value from 'n-1' elements (from 1 to n) of the array and then swaps it with the second element of the array. The second pass takes 'n-2' comparisons and so on. Following table shows the number of comparisons in each pass of the selection sort algorithm:

Sr#	Number of Comparisons
Pass:1	n-1
Pass:2	n-2
Pass:3	n-3
-----	----
Pass:n-3	n-(n-3) = 3
Pass:n-2	n-(n-2) = 2
Pass:n-1	n-(n-1) = 1

$$\text{Total comparisons} = (n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1$$

It is the arithmetic series. The formula for this arithmetic series:

$$\begin{aligned} (n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1 &= \frac{(n-1)(1+n-1)}{2} \\ &= \frac{(n^2 - n)}{2} \\ &= \frac{n^2}{2} - \frac{n}{2} \end{aligned}$$

As n^2 has a higher growth rate. Hence, the time complexity of the selection sort algorithm is $O(n^2)$.

The space complexity of the selection sort algorithm is $O(1)$ because only a single additional memory space is required i.e. for the 'temp' variable. Following table shows the time complexity and space complexity of the selection sort algorithm:

Time Complexity			Space Complexity
Best-Case	Average-Case	Worst-Case	Worst-Case
$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$

8.2.3 Insertion Sort

The insertion sort works by inserting each item into its proper position in the list. This sort method is relatively simple and easier to implement. However, it is inefficient for larger lists.

In insertion sorting, the list of the array is read from the beginning to the end. After each iteration, one element is inserted into its correct position relative to the previously sorted elements of the list. The array elements are not swapped or interchanged. These are shifted towards the right of the list to make room for the element to be inserted.

Example:

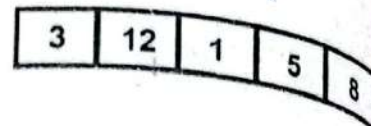
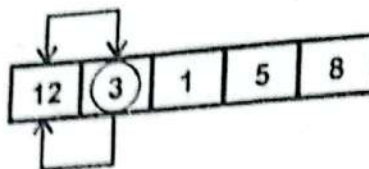
Suppose A is an array with 5 elements $A[0]$, $A[1]$, $A[2]$, $A[3]$, $A[4]$ having values 12, 3, 1, 5, 8 respectively as shown below;

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
12	3	1	5	8

To sort this array in ascending order, 4 passes or iterations are required. These are explained below:

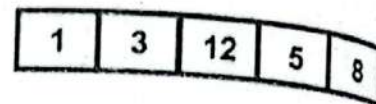
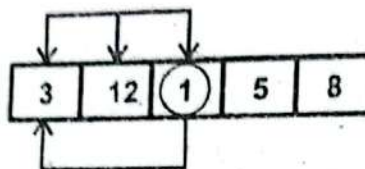
Pass-1:

Compare the value of the second element of the array with the value of the element before it and insert it in the proper position. In this case, the value of $A[1]$ is smaller than the value of $A[0]$. So shift the value of $A[0]$ one position to the right i.e. at position $A[1]$ and insert the value of $A[1]$ into $A[0]$. The array, before and after pass-1 is shown below:



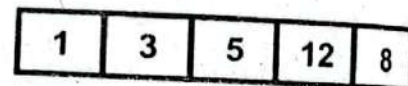
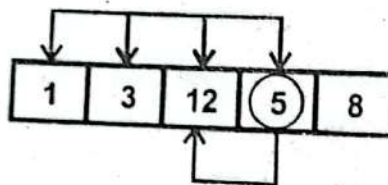
Pass-2:

Compare the value of the third element of the array with the values of elements before it and insert it in the proper position. In this case, the value of $A[2]$ is smaller than the value of $A[0]$. So shift the values of elements from $A[1]$ to $A[0]$ one position to right and insert the value of $A[2]$ into $A[0]$. The array, before and after pass-2 is shown below:



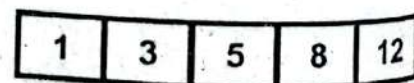
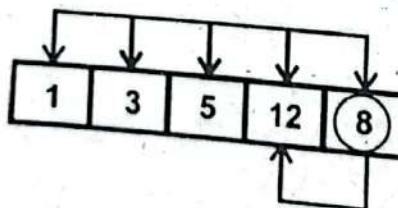
Pass-3:

Compare the value of the fourth element of the array with the values of elements before it and insert it in the proper position. In this case, the value of $A[3]$ is smaller than the value of $A[2]$. So shift the value of element from $A[2]$ one position to the right and insert the value of $A[3]$ into $A[2]$. The array, before and after pass-3 is shown below:



Pass-4:

Compare the value of the fifth element of the array with the values of elements before it and insert it in the proper position. In this case, the value of $A[4]$ is smaller than the value of $A[3]$. So shift the value of element from $A[3]$ one position to the right and insert the value of $A[4]$ into $A[3]$. The array, before and after pass-4 is shown below:



Algorithm - Insertion Sort

Write an algorithm to sort an array 'ABC' in ascending order using the insertion sort method. The array consists of N elements.

1. START
2. INPUT data into array ABC
3. [Start of upper Loop]
REPEAT Step-4 TO 8 FOR $U = 1$ TO $N-1$

4. VAL = ABC[U]
5. I = U
6. [Start of inner Loop]
REPEAT Step-7 WHILE (I >= 0 AND VAL < ABC[I-1])
7. ABC[I] = ABC[I - 1] [Shift to right]
I = I - 1
[End of step-6 inner loop]
8. ABC[I] = VAL [Insert value]
[End of step-3 upper loop]
9. EXIT

Program Code 8.9

// Program to sort an array with 5 elements by using the insertion sort method

```
#include <conio.h>
```

```
#include <iostream.h>
```

```
class Insertion
```

```
{
```

```
private:
```

```
int abc[5];
```

```
public:
```

```
void input(void);
```

```
void sort(void);
```

```
void print(void);
```

```
};
```

```
int
```

```
void main(void)
```

```
{
```

```
Insertion obj;
```

```
clrscr();
```

```
obj.input();
```

```
obj.sort();
```

```
obj.print();
```

```
getch();
```

```
} //end of main()
```

```
// Member function to input data into array
```

```
void Insertion::input(void)
```

```
{
```

```
cout<<"Enter 5 values "<<endl;
```

```
for(int i = 0; i<=4; i++)
```

```
cin>>abc[i];
```

```
} //end of input()
```

```
// Member function to sort data of array
```



```

void insertion::sort(void)
{
    int val, i, u;
    for(u = 1; u <= 4; u++)
    {
        val = abc[u];
        for(i = u; i >= 0 && val < abc[i - 1]; i--)
            abc[i] = abc[i - 1];    // Shift to right
        abc[i] = val;    // Insert value
    }
} //end of sort()

// Member function to print data of array
void insertion::print(void)
{
    cout << "Sorted array " << endl;
    for(int i = 0; i <= 4; i++)
        cout << abc[i] << " ";
    cout << endl;
} //end of print()

```

Output of the program:

Enter 5 values

25

42

9

1

2

Sorted array

1

2

9

25

42

8.2.3.1 Complexity Analysis of Insertion Sort Algorithm

The insertion sort algorithm uses a nested loop. The outer loop is used to control the number of passes. The inner loop is used for comparisons and shifting values (if needed) towards the right. Suppose we have an unsorted array of size 'n'. The number of passes required to sort this array will be 'n-1'.

In the first pass, it selects a value from the second element of the array and then compares it with the value of the first element of the array. If the selected value is less than the value of the first element, the value of the first element is shifted toward the right i.e. in the second element, and the value of the second element is shifted in the first element of the array using swapping. The first pass takes only '1' comparison.

In the second pass, it selects a value from the third element of the array and then compares it with the values of the second and first elements of the array one by one. The values

of the second and first elements are shifted toward the right and the value of the third element is shifted toward the right in the proper location (if needed). The second pass takes '2' comparison and so on.

Following table shows the number of comparisons in each pass of the insertion sort algorithm:

Sr#	Number of Comparisons
Pass:1	1
Pass:2	2
Pass:3	3
-----	----
Pass:n-3	(n-3)
Pass:n-2	(n-2)
Pass:n-1	(n-1)

$$\text{Total comparisons} = 1 + 2 + 3 + \dots + (n-3) + (n-2) + (n-1)$$

It is the arithmetic series. The formula for this arithmetic series:

$$\begin{aligned}
 1 + 2 + 3 + \dots + (n-3) + (n-2) + (n-1) &= \frac{(n-1)(1+n-1)}{2} \\
 &= \frac{(n^2 - n)}{2} \\
 &= \frac{n^2}{2} - \frac{n}{2}
 \end{aligned}$$

As n^2 has a higher growth rate. Hence, the time complexity of the insertion sort algorithm is $O(n^2)$.

The space complexity of the insertion sort algorithm is $O(1)$ because only a single additional memory space is required i.e. for the 'value' variable. Following table shows the time complexity and space complexity of the insertion sort algorithm:

Time Complexity			Space Complexity
Best-Case	Average-Case	Worst-Case	Worst-Case
$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$

8.2.4 Shell Sort

Shell sort method was discovered by Donald Shell in 1959. It is named in the honour of its inventor. It is also known as *Shell's method* and is one of the oldest sorting method. It is an improved form of *insertion sort* method (or variation of insertion sort). It works by comparing elements that are a certain distance away from each other (i.e. separated by a gap of different

positions) rather than adjacent elements in the array. It sorts the array or list in different phases or passes. The distance or gap size decreases in each phase until the gap reaches 1. Thus in the last phase, the adjacent elements are compared. It means that Shell sort works as insertion sort in the last phase.

Note that in insertion sort the gap between comparing elements is 1. Therefore, if gap = 1, then the Shell sort method becomes like the insertion sort method. The running time of the Shell sort is dependent on the gap sequence it uses.

Shell sort is an in-place sorting method as it requires no additional space in memory during execution. It is fast, easy to understand, and easy to implement. It is 5 times faster than bubble sort and twice faster than insertion sort. However, its complexity analysis tells us that it is complicated.

Determining Gap

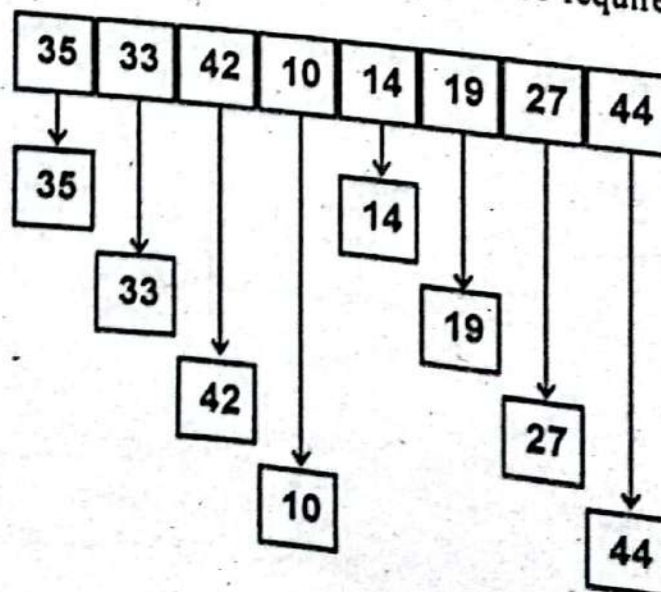
The fundamental problem of the Shell sort is to determine the optimal gap between the compared elements. Donald Shell proposed an initial gap of size $N/2$, where N is the size of the array. The initial gap is divided by 2 in each phase until the gap becomes 1. This approach has one major disadvantage that elements at odd and even locations of the array are mutually compared only in the last step.

Example:

An array with 8 elements and filled with data values is shown in the following figure. Assume that the name of the array is 'A'.

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
35	33	42	10	14	19	27	44

The value of the initial gap is 4 ($8/2$, where 8 is the size of the array). To sort this array in ascending order using the Shell sort method, 3 phases will be required. These are described as follows:



Phase -1:

In the first phase, the gap is 4. So the elements of the array with a gap of 4 are compared. These are compared in the following order:

$\frac{n}{2}$ → number by array

- A[0] is compared with element A[4]; as 35 is greater than 14, the values are interchanged. The new status of the array will be:

14	33	42	10	35	19	27	44
----	----	----	----	----	----	----	----

- A[1] is compared with element A[5]; as 33 is greater than 19, the values are interchanged. The new status of the array will be:

14	19	42	10	35	33	27	44
----	----	----	----	----	----	----	----

- A[2] is compared with element A[6]; as 42 is greater than 27, the values are interchanged. The new status of the array will be:

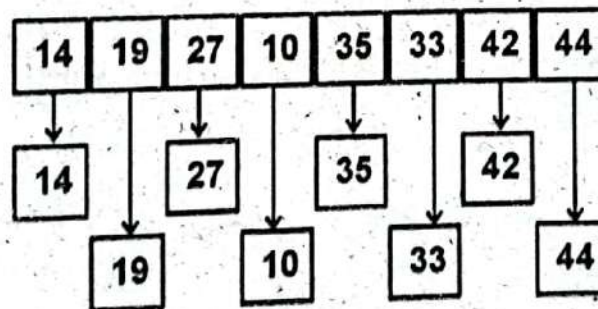
14	19	27	10	35	33	42	44
----	----	----	----	----	----	----	----

- A[3] is compared with element A[7]; as 10 is smaller than 44, the values are not interchanged. Thus the status of the array after the first phase will be:

14	19	27	10	35	33	42	44
----	----	----	----	----	----	----	----

Phase -2:

In the second phase, the gap is 2 ($4/2 = 2$). So the elements of the array with a gap of 2 are compared. These are compared in the following order:



- A[0] is compared with element A[2]; as 14 is smaller than 27, the values are not interchanged.

14	19	27	10	35	33	42	44
----	----	----	----	----	----	----	----

- A[4] is compared with element A[6]; as 35 is smaller than 42, the values are not interchanged.

14	19	27	10	35	33	42	44
----	----	----	----	----	----	----	----

- A[1] is compared with element A[3]; as 19 is greater than 10, the values are interchanged.

14	10	27	19	35	33	42	44
----	----	----	----	----	----	----	----

- A[5] is compared with element A[7]; as 33 is smaller than 44, the values are not interchanged.

14	10	27	19	35	33	42	44
----	----	----	----	----	----	----	----

Phase -3:

In the third phase, the gap is 1 ($2/2 = 1$). So the elements of the array with a gap of 1 are compared. It is the last phase. In this phase, Shell sort works like insertion sort. After this phase, the array is sorted completely.

10	14	19	27	33	35	42	44
----	----	----	----	----	----	----	----

Algorithm – Shell Sort

Write an algorithm to sort an array 'ARR' in ascending order using the Shell sort method. Assume that the array consists of N elements.

1. START
2. INPUT data into array ARR
3. GAP = N/2
4. REPEAT step-5 to step-10 WHILE GAP > 0
5. REPEAT Step-6 TO 9 FOR I = GAP TO N
6. TEMP = ARR[I]
7. REPEAT Step-8 WHILE (J >= I AND TEMP < ARR[J-GAP])
8. ARR[J] = ARR[J-GAP] [Shift to right]
9. ARR[J] = TEMP [Insert value]
10. GAP = GAP/2
11. PRINT sorted data of the array
12. EXIT

Program Code 8.10

```
// Program to sort array with 5 elements by using Shell sort method
#include <conio.h>
#include <iostream.h>

class Shell
```

```
{
    private:
        int arr[5];           // An array of integers
    public:
        void input(void);
        void sort(void);
        void display(void);
};

void main(void)
{
    Shell obj;
    clrscr();
    obj.Input();
    obj.sort();
    obj.display();
    getch();
} // end of main()

// Member function to input data into the array
void Shell::Input(void)
{
    cout<<"Enter 5 values "<<endl;
    for(int i = 0; i<=4; i++)
        cin>>arr[i];
} // end of input()

// Member function to sort data of the array
void Shell::sort(void)
{
    int j, gap;
    gap = 5/2;
    while(gap > 0)
    {
        for (int i = gap; i < 5; i++)
        {
            int temp = arr[i];
            for(j = i; j>=gap && temp < arr[j-gap]; j -= gap)
                arr[j] = arr[j - gap];
            arr[j] = temp;
        }
        gap = gap/2;
    }
} // end of Input()

// Member function to display data of the array
void Shell::display(void)
```



```

{
    cout<<"Sorted array "<<endl;
    for(int i = 0; i<=4; i++)
        cout<<arr[i]<<"\t";
    cout<<endl;
} // end of display()

```

Output of the program:

Enter 5 values

66

2

1

99

3

Sorted array

1

2

3

66

99

8.2.4.1 Complexity Analysis of Shell Sort Algorithm

The time complexity of the Shell sort algorithm depends on a gap sequence it uses. It also uses three loops in the nested form. The outer-most while loop is used to control the gap sequence. The gap of the Shell sort is repeatedly divided by 2 in each iteration. Suppose we have an unsorted array of size 'n'. In the first iteration of the outer while, the gap will be $n/2$, in the second iteration, the gap will be $n/4$, and so on. In the last iteration, the gap will be 1. Therefore, the time complexity for the upper loop will be:

$$= n/2 + n/4 + \dots + 1$$

It is the geometric series. Solving it, we have $O(2n)$. As 'n' has a higher growth rate than constants. Hence, the time complexity for the upper loop is $O(n)$.

The inner 'for' loop swaps the elements of the array based on the gap. The inner loop takes time which is equal to $O(n \log^2 n)$. Therefore, the time for the whole algorithm is the sum of the times for both loops:

$$O(n) + O(n \log^2 n) = O(n + n \log^2 n) = O(n \log^2 n)$$

Therefore, the overall worst-case time complexity of the Shell sort algorithm is $O(n \log^2 n)$.

The Shell sort algorithm performs swap operation on the input array. Therefore, the total space complexity of the Shell sort algorithm is also $O(1)$. Following table shows the time complexity and space complexity of the Shell sort algorithm:

Time Complexity			Space Complexity
Best-Case	Average-Case	Worst-Case	Worst-Case
$\Omega(n \log n)$	$\Theta(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$

8.2.5 Quick Sort

Quick sort is also known as *partition exchange sort*. As its name implies, the Quick sort is a very fast sorting method. It sorts any list or array very quickly. It is based on the *divide and conquer* rule (i.e. it is based on the partitioning of the array into smaller arrays). It works by partitioning the array to be sorted, then recursively sort each partition.

The basic concept of the Quick sort process is to pick up one element from an array and rearrange the remaining elements around it. This element logically divides the main array into two sub-arrays. The chosen element is called a *pivot* or *pivot element*. It is used as *split-point* (The location in the array where the array is divided is called *split-point*). We can choose the first, middle, last, or any random element of the array as a pivot element.

Quick sort method, divides the array or sub-array into three main parts in each phase.

- i) Elements having values smaller than the pivot element
- ii) Pivot element
- iii) Elements having values greater than the pivot element

Working of Quick Sort

Quick sort works as follows:

- 1- A pivot element is chosen from the array.
- 2- The array is reordered or rearranged. All the values smaller than the pivot are moved to the left side of the pivot element and all the values greater than the pivot are moved to the right side of the pivot element (with values equal to the pivot going either side). The array is logically divided into two sub-arrays.

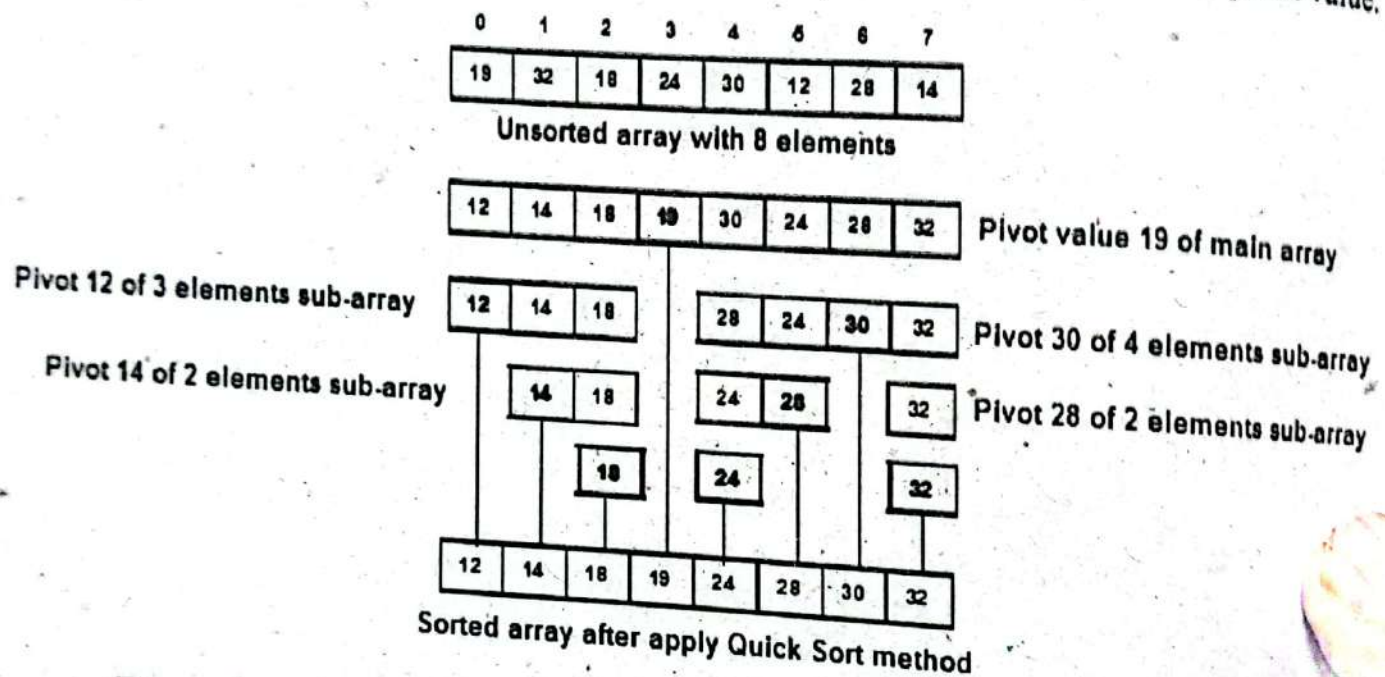
In this step, data stored in an array is read (or examined) from the left side and then right side at a time and then moved to the proper sides of the pivot element. In this process, two pointers work together, namely L (for the left marker) and R (for the right marker). At the beginning of the partition, L points to the first element of the array and R points to the last element of the array. The pointer L moves forward, while the pointer R moves backward.

Following steps are performed for shifting smaller and larger data values around the pivot element:

- i) The pointer, L moves forward (by increasing its value) until the value of L is greater than or equal to R. The value of each element is compared with the pivot element. When any element has greater or equal value to the pivot element, then the reading process stops at that point and the reading process begins from the right side of the array. The pointer R moves backward (by decreasing its value) until the value of R is smaller than or equal to L. The value of each element is compared with the pivot element. When any element has less or equal value to the pivot element, then the reading process stops at that point.

- ii) Swap the values of elements of L and R locations. After this, increase the value of L (i.e. $L = L + 1$) and decrease the value of R (i.e. $R = R - 1$).
 - iii) The step-(i) and step-(ii) are repeated until the value of R becomes smaller than or equal to L. At this point, note the location of R which will be the split-point.
 - iv) Swap the value of the pivot element with the split-point (i.e. location of R). The pivot shifts to its final position. Thus, the array is logically divided into two sub-arrays or sub-lists.
- 3- The step-1 and step-2 are repeated for each sub-array (applied recursively for each sub-array) until the entire array is sorted.

Following figure shows the Quick sort processing of unsorted array having 8 elements. In this sort processing, the first element of the array and sub-arrays is selected as the pivot value.



Example:

An array with 10 elements filled with data values is shown in the following figure. Assume that the name of the array is 'A'.

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
5	7	1	2	10	4	9	3	6	8

Following phases will be required to sort this array in ascending order using the Quick sort method:

Phase -1:

In this phase, the main array is logically divided into two sub-arrays. Following steps are performed in the first phase. The pointer L is at A[0] and R is at A[9]:

- 1- Choose the pivot element. Suppose we choose the first element $A[0]$ as the pivot element. Its value is 5.
- 2- Move the pointer L forward. The value of $A[1]$ is greater than the pivot element (i.e. $7 > 5$), so stop pointer L at index 1. Now move pointer R backward. The values of $A[9]$ and $A[8]$ are greater than the pivot element but the value of $A[7]$ is smaller than the pivot element (i.e. $3 < 5$), so stop pointer R at index 7. Swap the values of $A[1]$ and $A[7]$ and then increase the value of L and decrease the value of R . Now the value of L is 2 and R is 6. The array after swapping becomes as follows:

5	3	1	2	10	4	9	7	6	8
---	---	---	---	----	---	---	---	---	---

- 3- Move the pointer L from index 2 to forward. The values of $A[2]$ and $A[3]$ are smaller than the pivot element but the value of $A[4]$ is greater than the pivot element (i.e. $10 > 5$), so stop pointer L at index 4. Now move pointer R backward from index 6. The value of $A[6]$ is greater than the pivot element but the value of $A[5]$ is smaller than the pivot element (i.e. $4 < 5$), so stop pointer R at index 5. Swap the values of $A[4]$ and $A[5]$ and then increase the value of L and decrease the value of R . Now the value of L is 5 and R is 4. The array after swapping will be as follows:

5	3	1	2	4	10	9	7	6	8
---	---	---	---	---	----	---	---	---	---

- 4- Now the value of R is smaller than that of L . The split-point has reached. The location of R is the split-point. Exchange the value of the pivot element with the split-point. Array after swapping will be as follows:

4	3	1	2	5	10	9	7	6	8
---	---	---	---	---	----	---	---	---	---

Phase -2:

In this phase, the sub-array located on the left side of the pivot element is sorted recursively. The pointer L is at $A[0]$ and pointer R is at $A[3]$:

- 1- Choose the pivot element for sub-array $A[0]$ to $A[3]$. Suppose we choose the first element $A[0]$ as the pivot element. Its value is 4.
- 2- Move the pointer L forward. The values from $A[1]$ to the end of the array are smaller than the pivot element, therefore stop pointer L at the end of the array i.e. at index 3. The value of R is equal to L . The split-point has reached. Exchange the value of pivot element with the split-point i.e. location of R . Array after swapping will be as follows:

2	3	1	4	5	10	9	7	6	8
---	---	---	---	---	----	---	---	---	---

Quick sort of sub-array from elements $A[0]$ to $A[2]$. The pointer L is at $A[0]$ and pointer R is at $A[2]$:

- Choose the pivot element for sub-array A[0] to A[2]. Suppose we choose the first element A[0] as the pivot element. Its value is 2.
- Move the pointer L forward. The value of A[1] is greater than the pivot element (i.e. $3 > 2$), therefore stop pointer L at index 1. Now move pointer R backward. The value of A[2] is smaller than the pivot element (i.e. $2 < 1$), so stop pointer R at index 2. Swap the values of A[1] and A[2] and then increase the value of L and decrease the value of R. Now the value of L is 2 and R is 1. The array after swapping will be as follows:

2	1	3	4	5	10	9	7	6	8
---	---	---	---	---	----	---	---	---	---

- Now the value of R is smaller than that of L. The split-point has reached. Exchange the value of the pivot element with the split-point. Array after swapping is as follows:

1	2	3	4	5	10	9	7	6	8
---	---	---	---	---	----	---	---	---	---

Phase -3:

In this phase, the sub-array located on the right side of the pivot element is sorted recursively. The pointer L is at A[5] and pointer R is at A[9]:

- 1- Choose the pivot element for sub-array A[5] to A[9]. Suppose we choose first element A[5] as the pivot element. Its value is 10.
- 2- Move the pointer L forward. The values from A[6] to the end of the array are smaller than the pivot element, so stop pointer L at the end of the array i.e. at index 9. The value of R is equal to L. The split-point has reached. Exchange the value of the pivot element with the split-point. Array after swapping becomes as follows:

1	2	3	4	5	8	9	7	6	10
---	---	---	---	---	---	---	---	---	----

Quick sort of sub-array from elements A[5] to A[8]. The pointer L is at A[5] and pointer R is at A[8]:

- Choose the pivot element for sub-array A[5] to A[8]. Suppose we choose first element A[5] as the pivot element. Its value is 8.
- Move the pointer L forward. The value of A[6] is greater than the pivot (i.e. $9 > 8$), therefore stop pointer L at index 6. Now, move pointer R backward. The value of A[8] is smaller than the pivot element (i.e. $6 < 8$), hence stop pointer R at index 8. Swap the values of A[6] and A[8] and then increase the value of L and decrease the value of R. Now the value of L is 7 and R is also 7. The array after swapping will be as follows:

1	2	3	4	5	8	6	7	9	10
---	---	---	---	---	---	---	---	---	----

- Now the value of R is equal to that of L. The split-point has reached. Exchange the value of the pivot element with the split-point. Array after swapping will be as follows:

1	2	3	4	5	7	6	8	9	10
---	---	---	---	---	---	---	---	---	----

Quick sort of sub-array from elements A[5] to A[6]. The pointer L is at A[5] and pointer R is at A[6]:

- Choose the pivot element for sub-array A[5] to A[6]. Suppose we choose first element A[5] as the pivot element. Its value is 7.
- Move the pointer L from A[5] to forward, A[6] is the end of the array, so stop pointer L at index 6. The value of R is equal to L. The split-point has reached. Exchange the value of the pivot element with the split-point i.e. A[5] and A[6]. Array after swapping will be as follows:

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

The array is sorted in ascending order.

Algorithm – Quick Sort

Write an algorithm that sorts an array ARR consisting of N elements using the Quick sort.

1. START
2. Input data into array ARR
3. Choose a pivot value
4. Put all the values smaller than that of pivot on one side i.e. left side and all the values greater than that of pivot on the other side i.e. right side. The array is divided into two halves based on the pivot value.
5. Recursively sort each half of the array using the same procedure following step-3 and step-4 until the entire array is sorted.
6. EXIT

Program Code 8.11

// Program to sort an array having 7 elements using the Quick sort method.

```
#include <conio.h>
```

```
#include <iostream.h>
```

```
quicksort(int [], int , int );
```

```
int partition(int [], int , int );
```

```
void main(void)
```

```
{
```



```

int i, arr[] = {2, 15, 1, 61, 11, 47, 8};
clrscr();
cout<<"Array before sorting"<<endl;
for(i = 0; i<=6; i++)
    cout<<arr[i]<<"\t";
cout<<endl;
quicksort(arr, 0, 6);
cout<<"Array after sorting"<<endl;
for(i = 0; i<=6; i++)
    cout<<arr[i]<<"\t";
cout<<endl;
getch();
} // end of main()

// definition of quicksort function
quicksort(int arr[], int first, int last)
{
    int split_point;
    if(first < last)
    {
        split_point = partition(arr, first, last);

        //sort left and right
        quicksort(arr, first, split_point-1);
        quicksort(arr, split_point + 1, last);
    }
} // end of quicksort()

// definition of partition function
int partition(int arr[], int first, int last)
{
    int temp, L, R, pivot = arr[first];
    L = first + 1;
    R = last;
    while(1)
    {
        while(arr[L] <= pivot && L <= R)
            L++;
        while(arr[R] >= pivot && R >= L)
            R--;
        if(R < L)
        {
            temp = arr[first];
            arr[first] = arr[R];
            arr[R] = temp;
        }
    }
}

```

```

        break;
    }
    else
    {
        temp = arr[L];
        arr[L] = arr[R];
        arr[R] = temp;
        L++;
        R--;
    }
}
return R;
} // end of partition()

```

Output of the program:

Array before sorting

2 15 1 61 11 47 8

Array after sorting

1 2 8 11 15 47 61

Explanation of the code

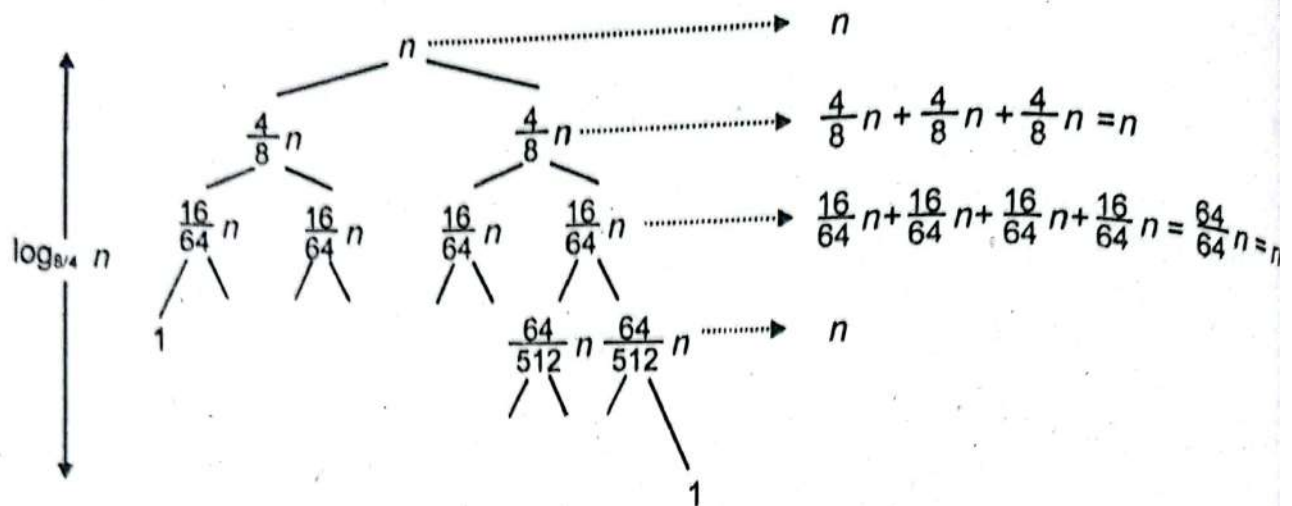
- **Call quicksort function:** Array of size 7 elements and parameters 0 and 6 are passed to the quicksort function for the first call.
- **Call partition function:** The partition function is called by passing an array, first and the last element as arguments or parameters. The 'first' parameter is the value of the starting index of array or sub-array, while the 'last' parameter is the value of the last index of the array.
- **Split_point:** In the partition function, we keep moving all the items smaller than the pivot value to the left and larger than the pivot value to the right. We have to keep track of the position of the partition so that we can split the array into two parts in the next step. The tracking of the index where we partition the array is done by using the split_point variable.

2.5.1 Complexity Analysis of Quick Sort Algorithm

Quick Sort is the recursive algorithm; therefore, it is analyzed using recursive methods of analysis. Consider the example given for the Quick sort having 8 elements of the input array. The algorithm picks 19 as the pivot element in its first step and divides the array into equal partitions. Equal partitions or division into two halves indicate the best-case of the Quick sort. The recursive function for this case will be written as:

$$= T(4n/8) + T(4n/8) + n$$

Where 8 indicates the total number of inputs, and 4 indicates the number of elements in the left and right partitions. The recursion tree for the above recursive function is shown below:



$$= n + n + n + n + \dots h = n \cdot h$$

Where 'h' is the height of the recursive tree which is equal to:

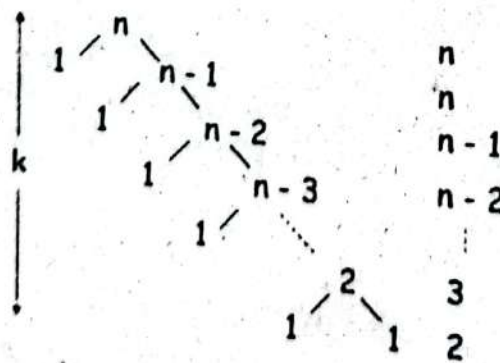
$$\log_{8/4} n = \log_2 n = \log n.$$

So, the time complexity of the Quick sort for the given example is $O(n \log n)$.

For the worst-case, the partition process either picks the smallest or greatest element as a pivot element from the input array. Hence, the recursive function for the worst-case scenario of the Quick sort is written as:

$$= T(1) + T(n-1) + n$$

Solving the above recurrence using the recursive tree method we have:



$$= n + n + (n-1) + (n-2) + (n-3) + \dots + 3 + 2$$

$$= n + [n + (n-1) + (n-2) + (n-3) + \dots + 3 + 2]$$

Where $[n + (n-1) + (n-2) + (n-3) + \dots + 3 + 2]$ is the arithmetic series from 2 to n. Put general form of arithmetic series - 1 in place of it, we have:

$$= n + [n(n+1)/2 - 1]$$

$$= n + [(n^2 - n) / 2]$$

$$= n + n^2/2 - n/2$$

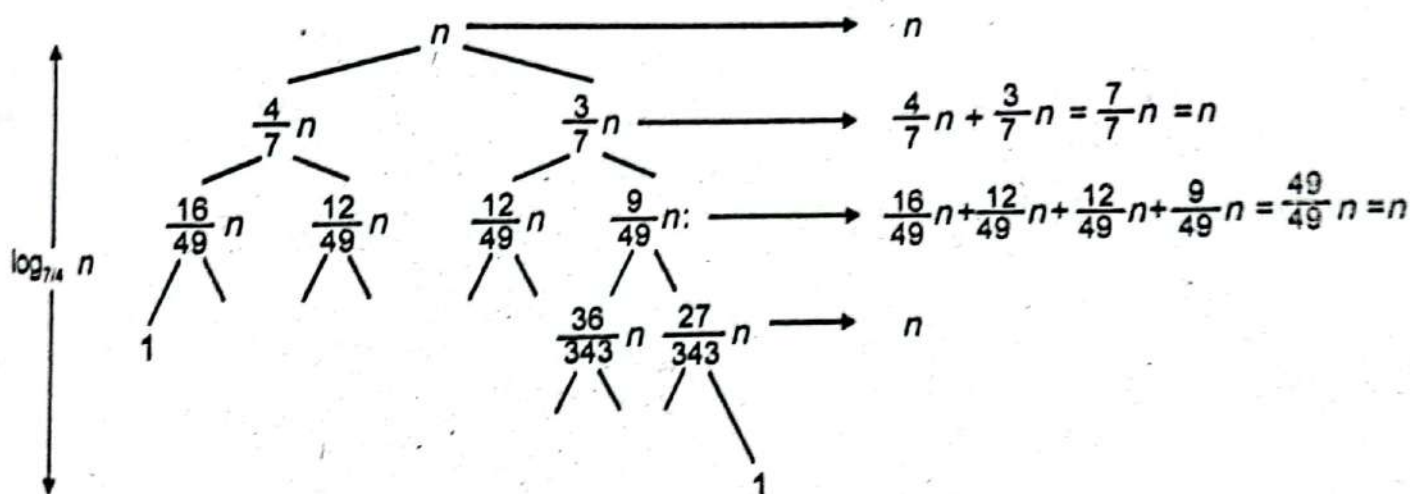
Where n^2 has the highest growth among all of them. Hence, the time complexity for the worst-case of quick sort is $O(n^2)$

For average case, when partitions are unequal in size, the recurrence function is written as:

$$T(a/n) + T(b/n) + n$$

Where n is the size of input, a is the left partition size, and b is the right partition size. The height of the recurrence tree in average-case is $\log n$, where base of the log is the reverse of the greater partition.

For example, if the recurrence function is: $T(4n/7) + T(3n/7) + n$, then the recurrence tree for the given recurrence function is shown below:



The partition $4/7$ is greater than partition $3/7$. So, the height of the recurrence tree is the reverse of the greater partition that is $7/4$. The time complexity for this recurrence function is $O(n \log_{7/4} n)$.

Quick sort algorithm has space usage of $O(\log n)$.

Following table shows the time complexity and space complexity of the Quick sort algorithm:

Time Complexity			Space Complexity
Best-Case	Average-Case	Worst-Case	Worst-Case
$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	$O(\log n)$

8.2.6 Merge Sort

Merge sort follows the rule of *Divide and Conquer*. It is a recursive sorting method that splits (divides) the unsorted list recursively into two halves until it cannot be further divided (i.e.

it divides the unsorted list into N sub-lists, each having one element). By definition, if there is only one element in the list, it is considered sorted. Following figure shows the splitting process of unsorted list into N sub-lists:

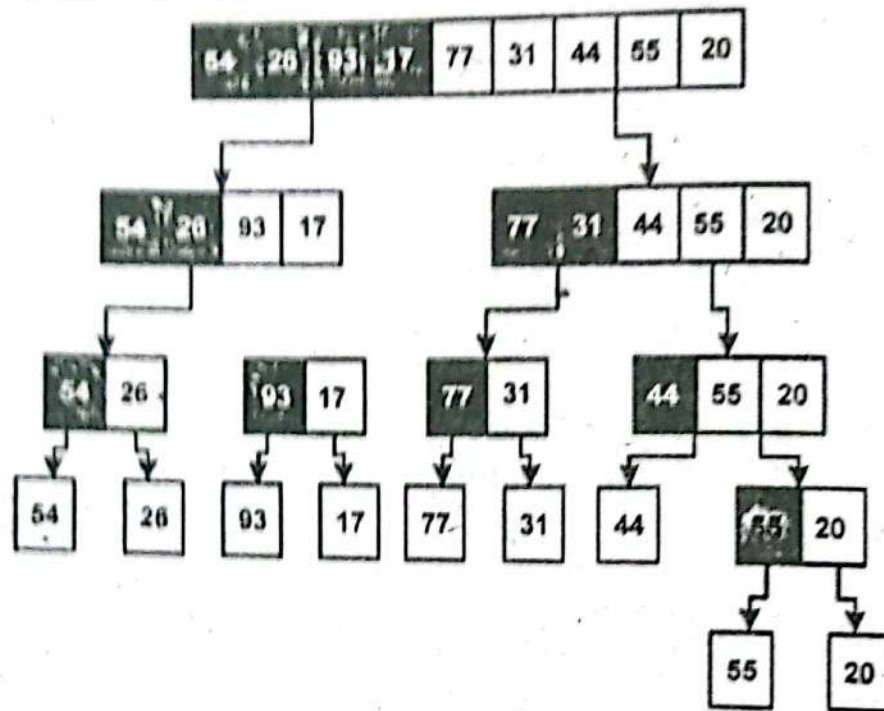


Figure: A splitting process of an unsorted list into N sub-lists

After splitting the unsorted list, the merging process is performed. In this process, smaller sub-lists are merged repeatedly into new sub-lists in sorted order, and at last one sorted list is produced. Following figure shows the merging process:

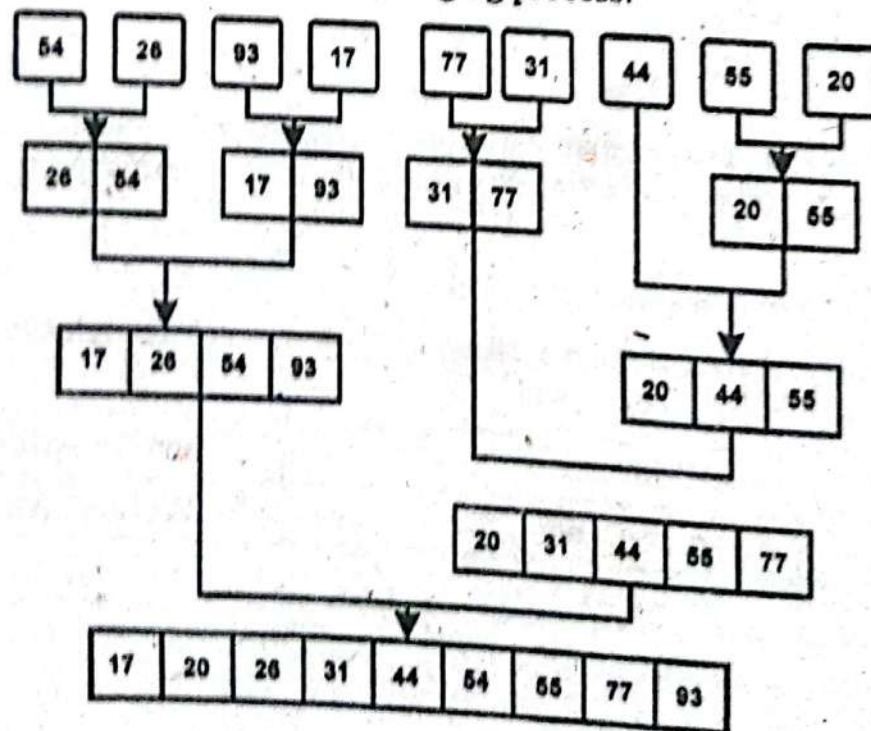


Figure: Merging process

How to Merge Sort Works?

To understand the working of the Merge sort, we take the following unsorted array of 8 elements:

15	34	29	11	36	17	41	45
----	----	----	----	----	----	----	----

Splitting Process

Merge sort first divides the whole array recursively into equal halves unless the atomic values are achieved. It does not change the sequence of the appearance of items in the original array. Following steps are performed to split or break the unsorted array into atomic elements:

1. In the first step, the above array of 8 elements is divided into two arrays, each of size 4.

15	34	29	11	36	17	41	45
----	----	----	----	----	----	----	----

2. In the second step, each array produced in step-1 is further divided into two arrays, each of size 2.

15	34	29	11	36	17	41	45
----	----	----	----	----	----	----	----

3. In the third step, each array produced in step-2 is further divided into two arrays, each of size 1. At this stage, we shall achieve atomic values that cannot be further divided.

15	34	29	11	36	17	41	45
----	----	----	----	----	----	----	----

Merging Process

Now, we combine sub-arrays in exactly the same manner as they were broken down. We first compare the element for each sub-array and then combine them into another array in a sorted manner. Following steps are performed in the merging process to merge the sub-arrays:

1. In the first step, we compare the two atomic values from left to right and change their order if required, for example:
 - We compare 15 and 34. These values are already in sorted positions, so we do not change their order.
 - We compare 29 and 11. The value of 29 is greater than 11, so we change their order i.e. put 11 first, followed by 29.
 - We compare 36 and 17. The value of 36 is greater than 17, so we change the order of 17 and 36.

- We compare 41 and 45. These values are already in sorted positions, so we do not change their order.

15	34	11	29	17	36	41	45
----	----	----	----	----	----	----	----

- 2- In the second step, we compare lists of two data values and merge them into another list of data values placing all in sorted order.

11	15	29	34	17	36	41	45
----	----	----	----	----	----	----	----

- 3- In the third step, we compare lists of four data values, and merge them into another list of data values. We get the sorted list of data values.

11	15	17	29	34	36	41	45
----	----	----	----	----	----	----	----

Algorithm – Merge Sorting

Write an algorithm that sorts an array ARR which consists of N elements using merge sort.

1. START
2. IF array ARR has only one element, it is already sorted, THEN
EXIT
ELSE
Divide the array recursively into two halves until it cannot be further divided
END IF
3. Merge the smaller sub-arrays into a new array in sorted order
4. EXIT

When an array of an odd number of elements is divided into two sub-arrays then the second sub-array will have one more element than the first sub-array. For example, the array with elements 8 is divided into two sub-arrays of equal lengths, i.e. each having 4 elements. However, if the length of an array is 9, then the first sub-array will have 4 elements and the second array will have 5 elements.

Program Code 8.12

```
//Program to sort array with 6 elements by using merge sorting method
#include <iostream.h>
#include <conio.h>
```

```
void MergeSort(int [], int, int);
void Merge(int [], int, int, int);
```

```
void main(void )
```

```

{
    int i, arr[] = {12, 11, 13, 5, 6, 7};
    clrscr();
    cout<<"The given array before sorting"<<endl;
    for(i = 0; i < 6; i++)
        cout<<arr[i]<<"\t";
    cout<<endl;
    MergeSort(arr, 0, 5);
    cout<<"Array after sorting"<<endl;
    for(i = 0; i < 6; i++)
        cout<<arr[i]<<"\t";
    getch();
} //end of main()

```

✓ // definition of MergeSort function that sorts sub-arrays

```

void MergeSort(int arr[], int start, int end)
{

```

```

    if (start < end)
    {
        int mid = (start+end)/2;
        // Sort first and second halves
        MergeSort(arr, start, mid);
        MergeSort(arr, mid+1, end);
        Merge(arr, start, mid, end);
    }
} //end of MergeSort()

```

✓ // definition of Merge function that merges the sub-arrays

```

void Merge(int arr[], int start, int mid, int end)
{

```

```

    int i, j, k;
    int n1 = mid - start + 1;
    int n2 = end - mid;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[start + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];
    // create temp arrays for left sub-array and right sub-array
    // copy data to temp array L[]
    // copy data to temp array R[]

```

```

    // Now merge the temp arrays back into arr[] in sorted order
    i = 0; // Initial index of first sub-array L
    j = 0; // Initial index of second sub-array R
    k = start; // Initial index of merged sub-array

```

```

    // merge data of L and R sub-arrays in sorted order
    while(i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i]; // copy from left sub-array
            i++;

```



```

    }
    else
    {
        arr[k] = R[j];    // copy from right sub-array
        j++;
    }
    k++;
}

// This loop copies remaining (if any) elements of L[] into arr[]
while(i < n1)
{
    arr[k] = L[i];
    i++;
    k++;
}

// This loop copies the remaining (if any) elements of R[] into arr[]
while(j < n2)
{
    arr[k] = R[j];
    j++;
    k++;
}
} //end of Merge()

```

Output of the program:

The given array before sorting

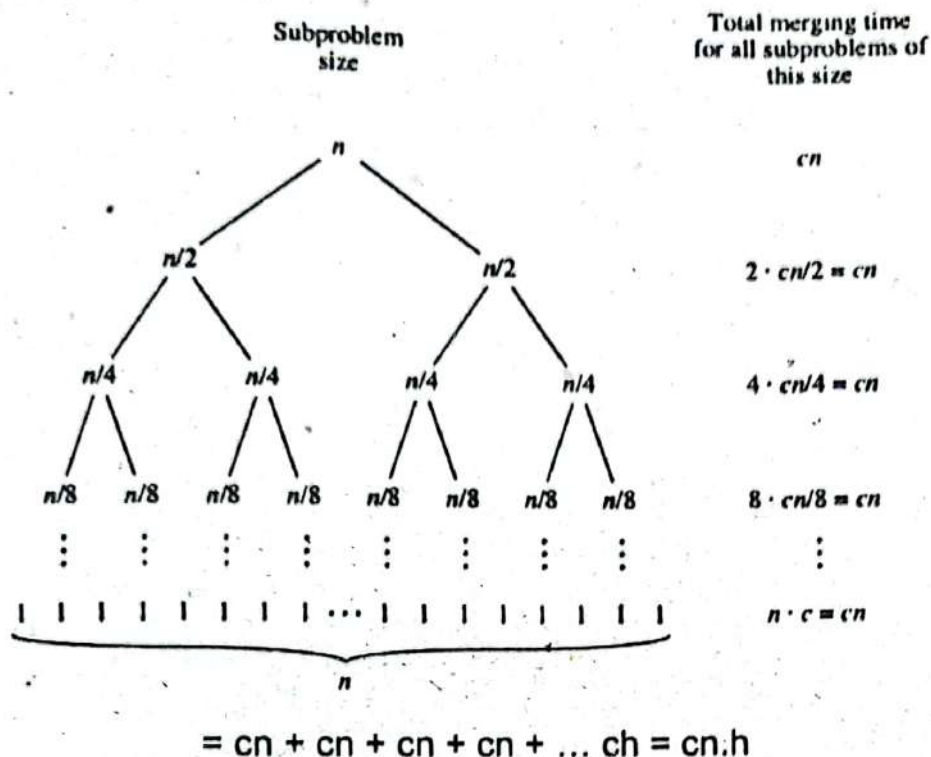
12	11	13	5	6	7
Array before sorting					
5	6	7	11	12	13

8.2.6.1 Complexity Analysis of Merge Sort Algorithm

Merge sort is also the recursive algorithm as Quick Sort; therefore, it is analyzed using recursive methods of analysis. The difference between the quick sort and merge sort is that the Quick sort can partition the input array into unequal sizes while merge sort always partitions into two halves. Hence, the analysis of the Merge sort is the same as the analysis of the best-case of the Quick sort. The recursive function for merge sort is as follows:

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \theta(n) & \text{if } n > 1 \end{cases}$$

The recursive tree for merge sort is shown below:



Where c is the constant and h is the height of the recursive tree which is equal to $\log n$. So, ignoring c , the time complexity of Merge Sort is $O(n \log n)$.

The space Complexity of the Merge Sort algorithm is $O(n)$. Following table shows the time complexity and space complexity of the Merge Sort algorithm:

Time Complexity			Space Complexity
Best-Case	Average-Case	Worst-Case	Worst-Case
$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$

8.2.7 Counting Sort

Counting sort is an integer sorting algorithm. It is used for sorting an array having small integer values. It does not involve a direct comparison between the elements of an array. It is based on keys between specific ranges. It works as follows:

- It counts the frequencies of elements of an array having distinct key values and stores them into an auxiliary array or count array.
- It maps the elements of the array to output sequence based on that auxiliary array.

This sorting technique is effective when the difference between different keys is not so big, otherwise, it can increase the space complexity.

Example:

Consider an array with 5 elements and filled with data values is shown in the following figure. Assume that the name of the array is 'A'.

A[0]	A[1]	A[2]	A[3]	A[4]
2	4	1	6	4

Following steps are performed to sort this array using the Counting sort method:

Step -1:

- Find the maximum value from the array A, which is 6.
- Declare an auxiliary array or count array C of int type with the size of maximum value found in array A and initialize each element of this array with a NULL value (i.e. $\text{int } A[6] = \{0\};$). The array C having 7 elements with index values 0 to 6 will be declared and value NULL or '0' will be assigned to all elements of A.

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	0	0	0	0	0	0

- Declare another array S with the same size as array C and initialize with a NULL value. This array will be used to store output i.e. sorted values.

S[0]	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]
-	-	-	-	-	-	-

Step -2:

Find the frequencies of elements of A and increment with 1 to the count array C such that: $C[A[\text{index}]] = C[A[\text{index}]] + 1;$

- A[0] is 2, so increment with 1 to the third element of C i.e. $C[2] = 0+1$. The new status of the array C will be:

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	0	1	0	0	0	0

- A[1] is 4, so increment with 1 to the fifth element of C i.e. $C[4] = 0+1$. The new status of the array C will be:

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	0	1	0	1	0	0

- A[2] is 1, so increment with 1 to the second element of C i.e. $C[1] = 0+1$. The new status of the array C will be:

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	1	1	0	1	0	0

- A[3] is 6, so increment with 1 to the seventh element of C i.e. $C[6] = 0 + 1$. The new status of the array C will be:

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	1	1	0	1	0	1

- A[4] is 4, so increment with 1 to the fifth element of C i.e. $C[4] = 1 + 1$. The new status of the array C will be:

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	1	1	0	2	0	1

Step -3:

Add consecutive elements of the array C, and update it with second element such that:
 $C[i+1] = C[i] + C[i+1]$

- $C[1] = C[0] + C[1] = 0 + 1 = 1$.

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
(0)	(1)	1	0	2	0	1

The new status of array C will be:

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	1	1	0	2	0	1

- $C[2] = C[1] + C[2] = 1 + 1 = 2$.

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	(1)	(1)	0	2	0	1

The new status of array C will be:

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	1	2	0	2	0	1

- $C[3] = C[2] + C[3] = 2 + 0 = 2$.

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	1	(2)	(0)	2	0	1

The new status of array C will be:

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	1	2	2	2	0	1

- $C[4] = C[3] + C[4] = 2 + 2 = 4$.

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	1	2	(2)	(2)	0	1

The new status of array C will be:

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	1	2	2	4	0	1

➤ $C[5] = C[4] + C[5] = 4 + 0 = 4.$

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	1	2	2	(4)	(0)	1

The new status of array C will be:

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	1	2	2	4	4	1

➤ $C[6] = C[5] + C[6] = 4 + 1 = 5.$

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	1	2	2	4	(4)	(1)

The new status of array C will be:

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	1	2	2	4	4	5

Step -4:

Pick a value from the first element of A and use this value of the element as an index element of array C. Now pick the value of this element of C and use this value of the element as an index element of array S. Copy the value of first element of array A into this element of S and decrement to $C[A[i]]$. Repeat this process from first element to the last element of the array A such as:

$$S[C[A[i]]] = A[i];$$

$$C[A[i]]--;$$

➤ A[0] is 1. $C[A[0]] = C[1] = 2$. So, $S[C[A[0]]] = S[C[1]] = S[2]$. Hence, copy A[0] to S[2]. The positions of arrays A, C and S after this steps are as follows:

A[0]	A[1]	A[2]	A[3]	A[4]
(2)	4	1	6	4

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	1	(2)	2	4	4	5

S[0]	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]
-	-	2	-	-	-	-

Decrement to $C[A[0]] = C[2] = 2 - 1 = 1$. Update Array C accordingly.

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	1	1	2	4	4	5

- > $A[1]$ is 4. $C[A[1]] = C[4] = 4$. So, $S[C[A[1]]] = S[C[4]] = S[4]$. Hence, copy $A[1]$ to $S[4]$. The positions of arrays A, C and S after this steps are as follows:

A[0]	A[1]	A[2]	A[3]	A[4]
2	4	1	6	4

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	1	1	2	4	4	5

S[0]	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]
-	-	2	-	4	-	-

Decrement to $C[A[1]] = 4 - 1 = 3$. Update Array C accordingly.

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	1	1	2	3	4	5

- > $A[2]$ is 1. $C[A[2]] = C[1] = 1$. So, $S[C[A[2]]] = S[C[1]] = S[1]$. Hence, copy $A[2]$ to $S[1]$. The positions of arrays A, C and S after this steps are as follows:

A[0]	A[1]	A[2]	A[3]	A[4]
2	4	1	6	4

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	1	1	2	3	4	5

S[0]	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]
-	1	2	-	4	-	-

Decrement to $C[A[2]] = 1 - 1 = 0$. Update Array C accordingly.

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	0	1	2	3	4	5

- > $A[3]$ is 6. $C[A[3]] = C[6] = 5$. So, $S[C[A[3]]] = S[C[6]] = S[5]$. Hence, copy $A[3]$ to $S[5]$. The positions of arrays A, C and S after this steps are as follows:

A[0]	A[1]	A[2]	A[3]	A[4]
2	4	1	6	4

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	0	1	2	3	4	5

S[0]	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]
-	1	2	-	4	6	-

Decrement to $C[A[3]] = 5 - 1 = 4$. Update Array C accordingly.

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	0	1	2	3	4	4

➤ A[4] is 4. $C[A[4]] = C[4] = 4$. So, $S[C[A[4]]] = S[C[4]] = S[4]$. Hence, copy A[4] to S[4]. The positions of arrays A, C and S after this steps are as follows:

A[0]	A[1]	A[2]	A[3]	A[4]
2	4	1	6	(4)

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	1	1	2	3	4	4

S[0]	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]
-	1	2	4	4	6	-

Decrement to $C[A[3]] = 3 - 1 = 2$. Update Array C accordingly.

C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]
0	0	1	2	2	4	4

Thus, the sorted is in array S, i.e.

S[0]	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]
-	1	2	4	4	6	-

Algorithm – Counting Sort

Write an algorithm to sort an array 'ARR' in ascending order using the Counting Sort method. Assume that the array consists of N elements.

1. START
2. INPUT data into array ARR within the range [0-N]
3. Find the maximum value from array ARR and store it into variable M.
4. Declare two array C and array S of size M and initialize them with value 0.
5. $I = 0$
6. Find the frequencies of the elements of the array ARR and add result in array C:
 $C[ARR[I]] = C[ARR[I]] + 1$
 $I = I + 1$
7. REPEAT step-6 for N-1 times
 [End of step-6 loop]
8. $I = 0$
9. Add consecutive elements of the array C, and update it with second element such that:
 $C[i+1] = C[i] + C[i+1]$
10. REPEAT Step-9 for M-1 times

10. REPEAT Step-9 for M-1 times
[End of-step-9 loop]
11. I = 0
12. S[C[A[I]]] = A[I]
13. C[A[I]] = C[A[I]] - 1
14. REPEAT Step-12 and Step 13 for N-1 times
[End of step-12 loop]
15. PRINT sorted data of the array
16. EXIT

Program Code 8.13

// Program to sort array with 5 elements within range [0-4] by using Counting sort method

```
#include <conio.h>
```

```
#include <iostream.h>
```

```
class Count
```

```
{
```

```
public:
```

```
void Count_Sort(int [], int);
```

```
int max_value(int []);
```

```
};
```

```
void main(void)
```

```
{
```

```
Count obj;
```



```

clrscr();
int m, ARR[5] = {2, 4, 1, 6, 4};
m = obj.max_value(ARR);
obj.Count_Sort(ARR, m);
getch();
} // end of main()

```

```

// Member function to sort data of the array
void Count::Count_Sort(int ARR1[], int mx)
{

```

```

    int *C, *S, i;
    C = new int [mx];           // create array C of size mx
    S = new int [mx];           // create array S of size mx
    for(i = 0; i <= mx; i++)    // This loop assigns NULL value into elements of arrays C & S
        C[i] = S[i] = NULL;

```

```

    // This loop finds the frequencies of the elements of the array ARR
    // and add result in array C:

```

```

    for(i = 0; i < 5; i++)
        C[ARR1[i]] = C[ARR1[i]] + 1;

```

```

    // This loop adds consecutive elements of the array C, and update it with second element

```

```

    for(i = 0; i < mx; i++)
        C[i+1] = C[i] + C[i+1];

```

```

    // This loop sorts the array ARR1 by copying the values of elements of ARR1
    // into proper locations in array S. The sorted data is available in array S

```

```

    for(i = 0; i <= 4; i++)
    {
        S[C[ARR1[i]]] = ARR1[i];
        C[ARR1[i]]--;
    }

```

```

    // This loop printed the sorted array S

```

```

    cout << "Sorted array " << endl;
    for(i = 0; i <= mx; i++)
        if(S[i] != NULL)
            cout << S[i] << "\t";

```

```

} //end of Count_Sort ()

```

```

// This function finds the maximum value from an array

```

```

int Count::max_value(int a[])
{

```

```

    int max = a[0];

```

```

for(int i=0; i<5; i++)
{
    if(a[i]>max)
        max = a[i];
}
return max;
} //end of max_value()

```

Output of the program:

Sorted array

1 2 4 4 6

8.2.7.1 Complexity Analysis of Counting Sort Algorithm

The counting sort algorithm uses only simple five for loops, without recursion or subroutine calls, it is straightforward to analyze.

- First for loop assigns NULL value to each element of arrays C and S. It takes $O(mx)$ time, where mx is the size of array C and S.
- Second for loop finds the frequencies of the elements of the array ARR and add result in array C. It takes $O(n)$ time, where n is the size of array ARR1.
- Third for loop performs consecutive elements of the array C. It takes $O(mx)$ time, where mx is the size of array C.
- Fourth for loop sorts the array ARR1 by copying the values of elements of ARR1 into proper locations in array S. It takes $O(n)$ time, where n is the size of array ARR1.
- Fifth for loop prints the sorted array S. It takes $O(mx)$ time, where mx is the size of array S.

Therefore, the time for the whole algorithm is the sum of the times for these steps:

$$\begin{aligned}
 O(mx) + O(n) + O(mx) + O(n) + O(mx) &= O(mx + n + mx + n + mx) \\
 &= O(2n + 3mx) \\
 &= O(n + mx)
 \end{aligned}$$

Therefore, the overall time complexity of counting sort algorithm is $O(n + mx)$

The counting sort algorithm uses arrays of length n and mx . Therefore, the total space usage of the algorithm is also $O(n + mx)$.

Following table shows the time complexity and space complexity of Counting Sort algorithm:

Time Complexity			Space Complexity
Best-Case	Average-Case	Worst-Case	Worst-Case
$\Omega(n + mx)$	$\Theta(n + mx)$	$O(n + mx)$	$O(n + mx)$

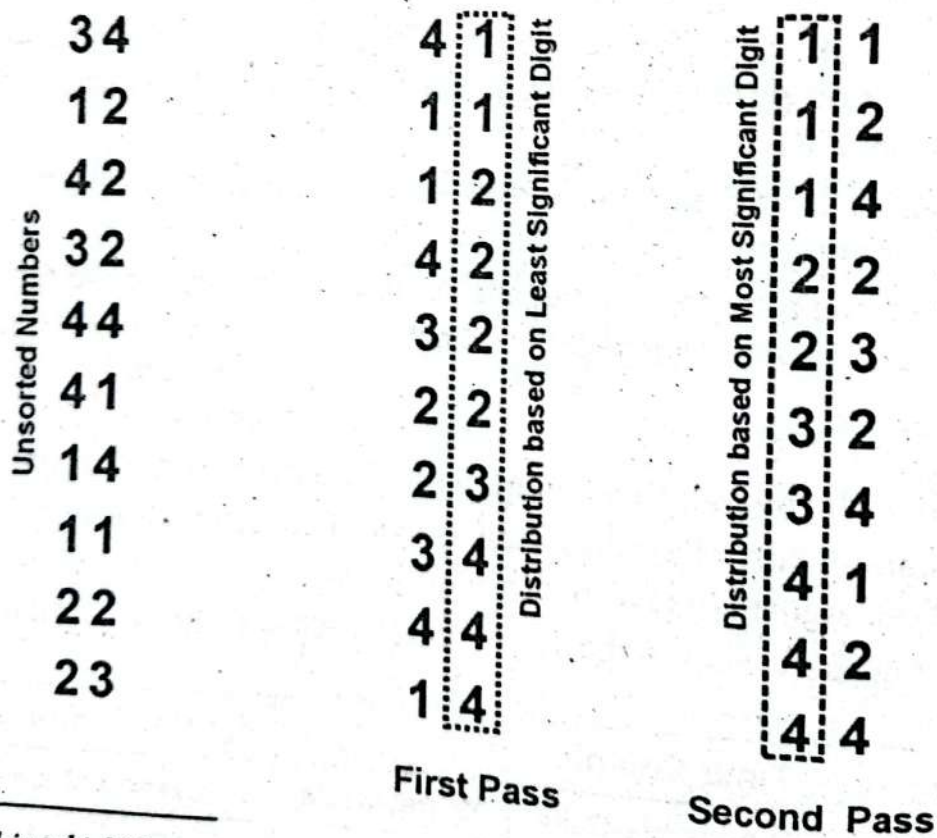
8.2.8 Radix Sort

Radix sort is the fastest sorting method. It is used for sorting large integers and strings. It treats an integer as a string of digits and arranges the integers in order by comparing the digits of the integers.

Radix sort is a non-comparison sorting method because it does not compare the numbers but distributes them into different groups based on their digits (significant digits i.e. least significant digit or most significant digit). Therefore, Radix sort method comes under the category of distribution sort*. It distributes data items to different *buckets* in the form of groups (buckets are storage area where data items with a common property are stored).

Radix sort method sorts the list of items (numbers or strings) in different phase. In first pass, numbers are distributed into groups based on the least significant digit. Similarly, in second pass numbers are redistributed into groups based on the next second least significant digit and so on. After each pass, numbers are collected from the buckets, keeping the numbers in order. When the last pass is done, numbers of array are sorted. The number of passes or iterations is based on length of the highest individual number in the list. Suppose the length of the highest number is 3 digits (i.e. 254), then 3 passes are required for sorting the list of numbers using Radix sort method.

Suppose the non-sorted values stored in an array are 34, 12, 42, 32, 44, 41, 14, 11, 22, and 23. Graphical representation of Radix sort for sorting this list of numbers of array is shown in the following figure:



* A sorting method in which data items are distributed to multiple places, then gathered and placed on the output in an order is called distribution sort.

Example:

Suppose an array of 10 elements filled with values is as follows:

134	312	242	432	244	141	114	211	222	323
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

In this array, values consist of four different digits (1, 2, 3, and 4). Therefore, four buckets are required for placing the groups of these values. Suppose these buckets are named as; B1, B2, B3, and B4. Further, the length of highest number is 3 digits, therefore 3 passes will be required.

Pass -1:

In first pass, following steps are performed:

- The values of array are distributed into groups based on the least significant digit and are placed into the relevant buckets. The number with least significant digit '1' will be placed in B1, number with least significant digit '2' will be placed in B2 and so on. For example, numbers 141 and 211 will be placed in B1, and numbers 312, 242, 432, and 222 will be placed in B2 and so on. After performing this step, values in the buckets will be distributed as;

B1	141	211				
B2	312	242	432	222		
B3	323					
B4	134	244	114			

- The numbers are collected from the buckets, keeping them in order and stored into the original array. New order of the numbers in the array will be;

141	211	312	242	432	222	323	134	244	114
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Pass -2:

In second pass, following steps are performed:

- The values of the array are redistributed into groups based on the next second least significant digit (in this case, next least significant digit is at mid position) and are placed in the relevant buckets. For example, numbers 211, 312 and 114 will be placed in B1, and numbers 222, and 323 will be placed in B2 and so on. After performing this step, the values in the buckets will be distributed as;

B1	211	312	114			
B2	222	323				
B3	432	134				
B4	141	242	244			

- The numbers are collected from the buckets, keeping them in order and stored into the original array. New order of the numbers in the array will be;

211	312	114	222	323	432	134	141	242	244
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Pass -3:

In third pass, following steps are performed:

- The values of the array are redistributed into groups based on the next third least significant digit and are placed in the relevant buckets. In this case, most significant digits because, total length of the numbers in the list is 3-digits. For example, numbers 114, 134 and 141 will be placed in B1, and numbers 211, 222 and 242 will be placed in B2 and so on. After performing this step, the values in the buckets will be distributed as;

B1	<u>1</u> 14	<u>1</u> 34	<u>1</u> 41			
B2	<u>2</u> 11	<u>2</u> 22	<u>2</u> 42	<u>2</u> 44		
B3	<u>3</u> 12	<u>3</u> 23				
B4	<u>4</u> 32					

- The numbers are collected from the buckets, keeping them in order and stored into the original array. New order of the numbers in the array will be;

114	134	141	211	222	242	244	312	323	432
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

After this process, we can say that the array is sorted in ascending order.

Algorithm – Radix Sort

Write an algorithm that sorts an array ARR which consists of N elements using Radix sort.

1. START
2. Input data into array ARR
3. Search the largest number from the array.
4. Determine the number of digits of the largest number and store the result in variable TOTAL_PASS
5. REPEAT step-6 to step-7 FOR I = 1 to TOTAL_PASS
6. Distribute the numbers of array into groups and store them into buckets.
7. Collect the data items (i.e. numbers) from the buckets, keeping them in order and store into the original array.
8. PRINT the sorted array.
9. EXIT

Implementation of Radix Sort

Radix sort can be implemented using different ways. Usually, one linear array is used to store the list of numbers, one 2-dimensional array is used for buckets to store different groups of numbers, and one linear array is used for counter. Each element of the counter array holds the number of data items in each bucket. Suppose the name of the array that is used to store the list of numbers is "arr" (of 10 elements), name of 2-dimensional array (of 10 row and 10 columns) is "bucket", and name of counter array (of 10 elements) is "counter". These arrays with data are shown as under;

Array "arr" with data items

0	1	2	3	4	5	6	7	8	9
8134	2311	7242	9435	3247	5156	1814	4231	6282	4340

Array "bucket" with data items after first pass

	0	1	2	3	4	5	6	7	8	9
0	4340									
1	2311	4231								
2	7242	6282								
3										
4	8134	1814								
5	9435									
6	5156									
7	3247									
8										
9										

Array "counter" with data after first pass

0	1	2	3	4	5	6	7	8	9
1	2	2	0	2	1	1	1	0	0

We have to perform the following steps to implement Radix sort method:

- 1- First of all, enter numbers into the array. Search the largest number from the array and count its total number of digits. Following function searches the largest number from the array and returns its total number of digits:

```
int largest_digits(int arr[])
```



```

    {
        int i, largest = arr[0], digits = 0;
        for(i = 1; i <= 9; i++)
        {
            if(arr[i] > largest)
                largest = arr[i];
        }

        // this loop finds the total number of digits in the largest number
        while(largest > 0)
        {
            largest = largest / 10;
            digits++;
        }
        return digits;
    }

```

- 2- Sort the array values by distributing these values into groups and store them in the bucket array. Similarly, record the number of data items in each bucket (each row of bucket) into the relevant index of counter array. In the above array 'bucket', row 0 indicates the bucket 0. It contains only one value. So 1 is stored at the index 0 (first element) of the counter array. Similarly, row 1 indicates the bucket 1 of bucket array. It contains two values. So 2 is stored at the index 1 (second element) of the counter array and so on. Following function sorts the array values by distributing these values into the bucket array and records the number of data items in each bucket into the relevant index of the counter array:

```

void sort(int arr[], int total_pass)
{
    int bucket[10][10], counter[10];
    int i, j, k, index, pass, divisor = 1;

    // different passes for sorting data
    for(pass = 1; pass <= total_pass; pass++)
    {
        /* before each pass, this loop initializes 0 value to all the
           elements of the counter array */
        for(i = 0; i <= 9; i++)
            counter[i] = 0;

        /* In each pass, this loop distributes the values of array and
           stores them in buckets. It also records the number of items of
           each bucket into the relevant index of the counter array */
    }
}

```

```

        for(i = 0; i <= 9; i++)
        {
            index = (arr[i] / divisor) % 10;
            bucket[index][counter[index]] = arr[i];
            counter[index]++;
        }

        /* In each pass, this nested loop collects data items from
        buckets and stores them in order into array "arr" */

        i = 0;
        for(j = 0; j <= 9; j++)
        {
            for(k = 0; k < counter[j]; k++)
            {
                arr[i] = bucket[j][k];
                i++;
            }
        }
        divisor = divisor*10;
    } //end of pass loop
} //end of sort function

```

Program Code 8.14

// Program to sort an array of 10 elements by using Radix sort method

```
#include <conio.h>
```

```
#include <iostream.h>
```

```
class Radix_sort
{
```

```
    private:
```

```
        int arr[10];
```

```
    public:
```

```
        void input(void);
```

```
        int largest_digits(void);
```

```
        void sort(int);
```

```
        void display(void);
```

```
};
```

```
void main(void)
```

```
{
```

```
    Radix_sort obj;
```

```
    int total_pass;
```

```
    clrscr();
```



```

        obj.input();
        total_pass = obj.largest_digits();
        obj.sort(total_pass);
        obj.display();
        getch();
    } //end of main()

    // Member function to input data into array
    void Radix_sort::input(void)
    {
        cout<<"Enter 10 values "<<endl;
        for(int i = 0; i<=9; i++)
            cin>>arr[i];
    } //end of input()

    // Member function to find the largest number of array and its total number of digits
    int Radix_sort::largest_digits()
    {
        int i, largest = arr[0], digits = 0;
        for(i = 1; i<=9; i++)
        {
            if(arr[i] > largest)
                largest = arr[i];
        }

        // this loop finds the total number of digits in the largest number
        while(largest>0)
        {
            largest = largest / 10;
            digits++;
        }
        return digits;
    } //end of largest_digits()

    // Member function to sort data of array
    void Radix_sort::sort(int total_pass)
    {
        int bucket[10][10], counter[10];
        int i, j, k, index, pass, divisor = 1;
        // different passes for sorting data
        for(pass = 1; pass<=total_pass; pass++)
        {
            // before each pass, this loop initializes 0 value to all elements of counter array
            for(i = 0; i <= 9; i++)
                counter[i] = 0;

            /* In each pass , this loop distributes the values of array and stores them
            in buckets. It also records the number of items of each bucket into
            relevant index of counter array */

```

```

for(i = 0; i <= 9; i++)
{
    index = (arr[i] / divisor) % 10;
    bucket[index][counter[index]] = arr[i];
    counter[index]++;
}

/* In each pass, this nested loop collects data items from buckets and
store them in order into array "arr" */
i = 0;
for(j = 0; j <= 9; j++)
{
    for(k = 0; k < counter[j]; k++)
    {
        arr[i] = bucket[j][k];
        i++;
    }
    divisor = divisor * 10;
} // end of pass loop
} // end of sort function

// Member function to display data of array
void Radix_sort::display(void)
{
    cout<<"Sorted array "<<endl;
    for(int i = 0; i<=9; i++)
        cout<<arr[i]<<"\t";
    cout<<endl;
} //end of display()

```

Output of the program:

Enter 10 values

125
9942
97
11
12
876
123
875
11
456

Sorted array

11 11 12 97 123 125 456 875 876 9942

8.2.8.1 Complexity Analysis of Radix Sort Algorithm

The radix sort algorithm uses three for loops within a loop (nested loop), without recursion or subroutine calls.

- The outer-most *for* loop controls different passes of radix sort. It takes $O(n)$ time, where n is the total number of passes or total number of digits of the largest number.
- First *for* loop within outer-most *for* loop assigns 0 value to each element of counter array. As it is within an outer loop, so the complexity of outer loop is multiplied with its complexity. It takes $O(nk)$ time, where n is the time complexity of outer loop, and k is the size of array.
- Second *for* loop within outer-most *for* loop distributes the values of array and stores them in buckets. It also records the number of items of each bucket into relevant index of counter array. It takes $O(k)$ time, where k is the size of array. As it is the inner loop; hence the time of outer-most loop is multiplied with its time. The overall time complexity is $O(nk)$.
- Third *for* loop within outer-most *for* loop also contains a nested *for* loop. It collects data items from buckets and store them in order into array. It takes $O(nk)$ time.

The time for the whole algorithm is the sum of the times for these steps:

$$\begin{aligned} O(n) + O(nk) + O(nk) + O(nk) &= O(n + nk + nk + nk) \\ &= O(n + 3nk) \\ &= O(nk) \end{aligned}$$

Therefore, the overall time complexity of radix sort algorithm is $O(nk)$.

The Radix Sort algorithm uses 2-dimensional array of length $n \times k$ (In our program code $k = n$). Therefore, the total space usage of the algorithm is also $O(n+k)$.

Following table shows the time complexity and space complexity of Radix Sort algorithm:

Time Complexity			Space Complexity
Best-Case	Average-Case	Worst-Case	Worst-Case
$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n + k)$

8.2.9 Bucket Sort

Bucket sort is also known as *bin sort*. It is similar to Radix sort. Like Radix sort, it distributes numbers (data items) into different buckets. Each bucket is then sorted individually using any suitable sorting algorithm or by recursively applying the bucket sorting algorithm. Therefore, bucket sort also comes under the category of distribution sort. Bucket sort can be implemented with comparisons and therefore, can also be considered a comparison sort method. Bucket sort is mainly useful when the input is uniformly distributed over a range of values.

Working of Bucket Sort

Bucket sort works as follows:

1. Data items or numbers are distributed into different buckets. Different techniques can be used to distribute numbers in buckets.
2. Each non-empty bucket is sorted individually using any sorting method.
3. Data items are collected or gathered from the buckets in order, to get the final sorted list of data items (numbers).

Difference between Bucket Sort and Radix Sort

Although both bucket sort and Radix sort are distribution sort methods and sort the numbers by distributing them into buckets. The main differences between these two sort methods are as follows:

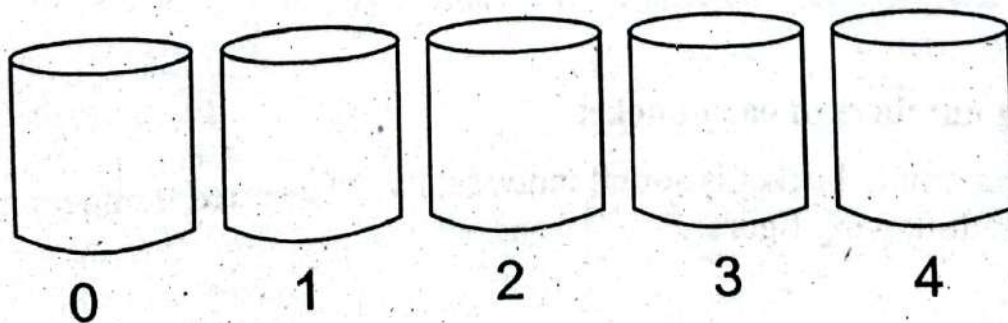
- Radix sort is non-comparison sorting method, while bucket sort is comparison sorting method because distributed numbers in buckets are mostly sorted using comparison sorting method like insertion sort, quick sort, or Shell sort etc.
- Radix sort distributes the list of numbers into buckets repeatedly, while bucket sort distributes the list of numbers into buckets only once and then sorts each bucket.
- Radix sort distributes the list of numbers into buckets based on the significant digits of the numbers, while bucket sort distributes the list of numbers using different techniques.

Example:

Suppose the unsorted list of numbers is as follows:

75 2 42 19 4 81 1

Suppose the buckets are 5 in which numbers are to be distributed. These are named as 0, 1, 2, 3 and 4 as shown in the following figure:



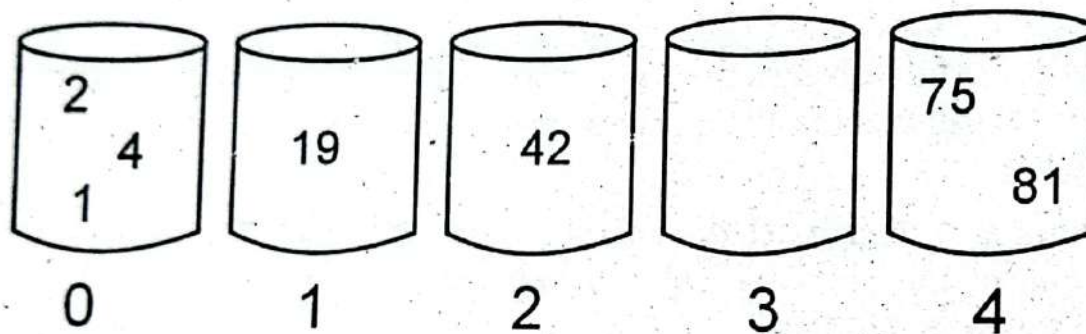
1- Distributing the numbers into buckets

Search the largest number to find the divider for distributing the numbers into buckets. In this list of numbers, the largest number is 81. So divider will be calculated as;

$$\begin{aligned} \text{ceil}((81+1)/5.0) &= \text{ceil}(82/5.0) \\ &= 17 \end{aligned}$$

So the above numbers will be distributed into five buckets as follows;

Number	Bucket	Description
75	4	$\text{floor}(75/17) = 4$
2	0	$\text{floor}(2/17) = 0$
42	2	$\text{floor}(42/17) = 2$
19	1	$\text{floor}(19/17) = 1$
4	0	$\text{floor}(4/17) = 0$
81	4	$\text{floor}(81/17) = 4$
1	0	$\text{floor}(1/17) = 0$



The `floor()` function returns the integral part of the float or double value that is not greater than the given value. It is smaller than or equal to the given value e.g.;

- If $x = 623.456$, then `floor(x)` will return 623.0
- If $x = -99.2$, then `floor(x)` will return -100.0

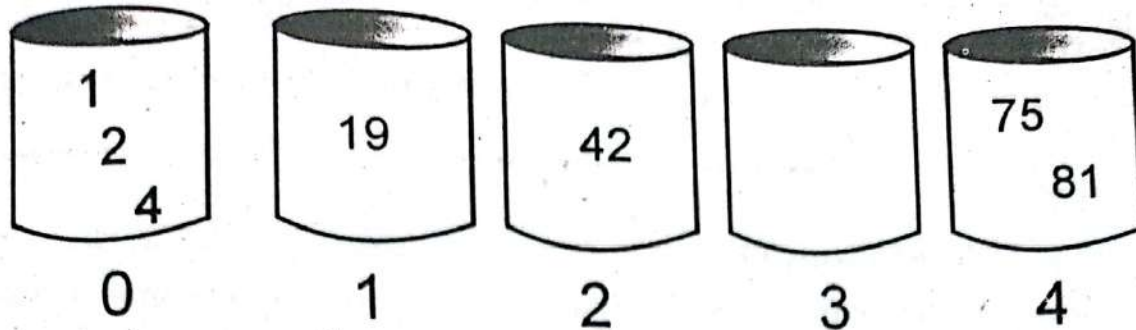


The `ceil()` function is similar to `floor()` function but `ceil()` function returns the integral part of the float or double value that is greater than the given value e.g.;

- If $x = 16.3$, then `ceil(x)` returns 17.0
- If $x = -19.3$, then `ceil(x)` returns -19.0

2- Sorting numbers of each bucket

Each non-empty bucket is sorted individually. The sorted numbers in each bucket are shown in the following figure:



3- Collecting numbers from each bucket in order

The numbers are collected from each bucket in order. The result will be the sorted numbers such as;

1 2 4 19 42 75 81

Implementation of Bucket Sort

Bucket sort can be implemented using different ways. Usually, one linear array is used to store the list of numbers, multiple linear arrays are used for buckets (or only one 2-dimensional array is used for buckets), and one linear array is used for counter. The number of data items in each bucket is recorded into the relevant index of counter array. Suppose the name of the array that is used to store the list of numbers is "arr" (of 7 elements), five linear arrays named B0, B1, B2, B3, and B4 (each of 7 elements), and name of counter array (of 5 elements) is "counter". These arrays with data are shown as under:

Unsorted data Array "arr"

0	1	2	3	4	5	6
75	2	42	19	4	81	1

Counter Array

0	1	2	3	4
3	1	1	0	2

The first element of counter array has value 3. This value indicates that B0 contains three data items. Similarly, 2nd and 3rd elements have value 1. It means that both buckets B1 and B3 store only one value and so on.

Bucket Arrays

Bucket "B0"

2	4	1				
---	---	---	--	--	--	--

Bucket "B1"

19						
----	--	--	--	--	--	--

Bucket "B2"

42						
----	--	--	--	--	--	--

Bucket "B3"

--	--	--	--	--	--	--

Bucket "B4"

75	81					
----	----	--	--	--	--	--

Algorithm – Bucket Sort

Write an algorithm that sorts an array ARR consisting of N elements using Bucket sort.

1. START
2. Input data into array ARR
3. Distribute the numbers of array into groups and store them into buckets.
4. Sort the numbers of each bucket.
5. Collect the numbers from the buckets, keeping them in order and store into the original array.
6. PRINT the sorted array.
7. EXIT

Program Code 8.15

// Program to sort an array of 10 elements by using Bucket sort method

```
#include <conio.h>
```

```
#include <iostream.h>
```

```
#include <math.h>
```

```
class bucket_sort  
{
```

```
private:
```

```
int arr[10];
```

```
int B0[10], B1[10], B2[10], B3[10], B4[10];
```

```
int counter[5];
```

```
public:
```

```
void input(void);
```

```
int largest(void);
```

```
void distribute_sort(int);
```

```
void sort(int [], int);
```

```
void concatenate(void);
```

```
void display(void);
```

```
};
```

```
void main(void)
```

```
{
```

```
    bucket_sort obj;
```

```
    int large;
```

```
    clrscr();
```

```
    obj.input();
```

```
    large = obj.largest();
```

```
    obj.distribute_sort(large);
```

```
    obj.concatenate();
```

```
    obj.display();
```

```
    getch();
```

```
} //end of main()
```

```

// Member function that inputs data into array
void bucket_sort::input(void)
{
    cout<<"Enter 10 integer values: "<<endl;
    for(int i = 0; i<=9; i++)
        cin>>arr[i];
} // end of input()

```

```

// Member function that finds the largest number of the array
int bucket_sort::largest()
{
    int i, max = arr[0];
    for(i = 1; i<=9; i++)
    {
        if(arr[i] > max)
            max = arr[i];
    }
    return max;
} // end of largest()

```

```

// Member function that distributes data array into buckets
void bucket_sort::distribute_sort(int large)
{
    int i, divider, res;
    for(i = 0; i<=4; i++)
        counter[i] = 0;
    divider = ceil((large+1) / 5.0);

    // this loop distributes numbers stored in array "arr" into five arrays
    // which are used as buckets
    for(i = 0; i <= 9; i++)
    {
        res = floor(arr[i] / divider);
        switch(res)
        {
            case 0:
                B0[counter[0]] = arr[i];
                counter[0]++;
                break;
            case 1:
                B1[counter[1]] = arr[i];
                counter[1]++;
                break;
            case 2:
                B2[counter[2]] = arr[i];
                counter[2]++;
                break;

```



```

// Member function that inputs data into array
void bucket_sort::input(void)
{
    cout<<"Enter 10 integer values: "<<endl;
    for(int i = 0; i<=9; i++)
        cin>>arr[i];
} // end of input()

```

```

// Member function that finds the largest number of the array
int bucket_sort::largest()
{
    int i, max = arr[0];
    for(i = 1; i<=9; i++)
    {
        if(arr[i] > max)
            max = arr[i];
    }
    return max;
} // end of largest()

```

```

// Member function that distributes data array into buckets
void bucket_sort::distribute_sort(int large)
{
    int i, divider, res;
    for(i = 0; i<=4; i++)
        counter[i] = 0;
    divider = ceil((large+1) / 5.0);

    // this loop distributes numbers stored in array "arr" into five arrays
    // which are used as buckets
    for(i = 0; i <= 9; i++)
    {
        res = floor(arr[i] / divider);
        switch(res)
        {
            case 0:
                B0[counter[0]] = arr[i];
                counter[0]++;
                break;
            case 1:
                B1[counter[1]] = arr[i];
                counter[1]++;
                break;
            case 2:
                B2[counter[2]] = arr[i];
                counter[2]++;
                break;
            case 3:
                B3[counter[3]] = arr[i];

```

```

        counter[3]++;
        break;
    case 4:
        B4[counter[4]] = arr[i];
        counter[4]++;
        break;
    } //end of switch
} // end of for loop

// these code sorts numbers stored into five buckets by calling sort() function
sort(B0, counter[0]);           // function call for sorting B0
sort(B1, counter[1]);           // function call for sorting B1
sort(B2, counter[2]);           // function call for sorting B2
sort(B3, counter[3]);           // function call for sorting B3
sort(B4, counter[4]);           // function call for sorting B4
} //end of distribute_sort function

```

// Member function to sort data of bucket using insertion sort method

```

void bucket_sort::sort(int B[], int size)
{

```

```

    int val, i, u;
    for(u = 0; u < size; u++)
    {
        val = B[u];
        for(i = u; i > 0 && val < B[i-1]; i--)
            B[i] = B[i-1];
        B[i] = val;
    }
} // end of sort()

```

// Member function that collects numbers from buckets in order
// and store in original array "arr"

```

void bucket_sort::concatenate(void)
{

```

```

    int i, c = 0;

    // Collect the numbers stored in bucket B0 (if any)
    for(i = 0; i < counter[0]; i++, c++)
        arr[c] = B0[i];

    // Collect the numbers stored in bucket B1 (if any)
    for(i = 0; i < counter[1]; i++, c++)
        arr[c] = B1[i];

    // Collect the numbers stored in bucket B2 (if any)
    for(i = 0; i < counter[2]; i++, c++)
        arr[c] = B2[i];

    // Collect the numbers stored in bucket B3 (if any)
    for(i = 0; i < counter[3]; i++, c++)

```



```

arr[c] = B3[i];

// Collect the numbers stored in bucket B4 (if any)
for(i = 0; i < counter[4]; i++, c++)
    arr[c] = B4[i];
} //end of concatenate()

// Member function to display data of array
void bucket_sort::display(void)
{
    cout << "Sorted array " << endl;
    for(int i = 0; i <= 9; i++)
        cout << arr[i] << " ";
    cout << endl;
} //end of display()

```

Output of the program:

Enter 10 integer values

122

12

90

11

942

876

123

875

9

455

Sorted array

9 11 12 90 122 123 455 875 876 942

8.2.9.1 Complexity Analysis of Bucket Sort Algorithm

The bucket sort algorithm uses two functions: `distribute_sort()` and `concatenate()`:

- The `distribute_sort` uses a 'for' loop to distribute input elements to the relevant buckets. It takes $O(n)$ time, where 'n' is the size of the array. Within this function, there is a call for sort function that sorts elements within a bucket for 'k' times where 'k' is the number of buckets. The insertion sort algorithm is used for sorting elements of the array; because for a small number of elements it gives results in constant time. Hence, the time complexity for `distribute_sort` function is $O(n) + O(1) = O(n)$.
- The `concatenate` function collects the number of buckets if any. It takes $O(k)$ time, where 'k' is the number of buckets.

Therefore, the time for the whole algorithm is the sum of the times for these steps:

$$O(n) + O(k) = O(n+k)$$

The average case time complexity for bucket sort is $\Theta(n + k)$ where 'n' is the length of the input sequence, and 'k' is the number of buckets. The problem is that its worst-case performance is $O(n^2)$ when the number of buckets is equal to the input size which makes it as slow as bubble sort.

The bucket sort algorithm uses an array of length 'n' and list of 'k' for buckets. Therefore, the total space usage of the algorithm is also $O(n+k)$.

Following table shows the time complexity and space complexity of the Bucket Sort algorithm:

Time Complexity			Space Complexity
Best-Case	Average-Case	Worst-Case	Worst-Case
$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n+k)$

Summary --- Complexities of Sorting Algorithms

Sorting Algorithm	Time Complexity			Space Complexity
	Best-Case	Average-Case	Worst-Case	Worst-Case
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Shell Sort	$\Omega(n \log n)$	$\Theta(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$
Quick Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	$O(\log n)$
Merge Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$
Counting Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(n+k)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n+k)$

Short Answers to the Questions

What is sequential search?

Sequential search is also known as *serial search* or *linear search*. It is a very simple and straight forward technique to search for a specified data item in an unordered list. The specified data item is searched in the list sequentially; i.e. starting from the first data item up to the required data item of the list in a sequence.

What is the difference between linear search and binary search?

Linear Search	Binary Search
A sorted list is not required.	A sorted list is required.
It can be used in linked list implementation.	It cannot be used in linked list implementation.

It is suitable for a list changing very frequently.	It is only suitable for static lists because any change requires resorting of the list.
The average number of comparisons is very high.	The average number of comparisons is relatively slow.

**What is bubble sort?**

Bubble sort method is the oldest and simplest sorting method. It is also known as *exchange sort*. It works by comparing two neighboring values of the list and swapping (exchanging) them if required.

**What is selection sort?**

Selection sort works by starting at the beginning of the array, comparing the first element with the other elements in the array. The smallest element is placed at position 0, and the sort then begins again at position 1. This continues until the entire array is sorted. Selection sort method is relatively easy to implement. It is also a faster method for sorting. It is, however, inefficient for large lists.

**Differentiate between bubble sort and selection sort algorithms.**

The main differences between bubble sort and selection sort algorithms are as follows:

- In the bubble sort, each element and its adjacent element is compared and swapped if required. On the other hand, selection sort works by selecting the element and swapping that particular element with the last element. The selected element could be the largest or smallest depending on the order i.e., ascending or descending.
- The worst-case complexity is the same in both the algorithms, i.e., $O(n^2)$, but best-case complexity is different. Bubble sort takes an order of 'n' time, whereas selection sort consumes an order of n^2 time.
- Bubble sort is a stable algorithm, while selection sort is unstable.
- Selection sort algorithm is fast, while the bubble sort is very slow.
- Bubble sort algorithm is considered to be the most simple and inefficient algorithm, but the selection sort algorithm is efficient than bubble sort.
- Bubble sort algorithm consumes additional space for storing temporary variable and needs more swaps than the selection sort algorithm.

**What is insertion sort?**

The insertion sort works by inserting each item into its proper position in the list. This sort method is relatively simple and easier to implement. However, it is inefficient for large lists.

**What is merge sort?**

Merge Sort works by breaking the list of data into two halves and recursively sorting each half. When the two halves are sorted, they are merged together to form the final sorted list.

EXERCISE

Q# 1 Select the correct answer from the given multiple choices.

1. The process of finding a specific data item or record from a given list is called:
 (a) Sorting (b) Finding (c) Searching (d) Locating
2. Sequential search is also called:
 (a) Serial search (b) Binary search (c) Step-wise search (d) None
3. Which one search method is used to find a specific data in a sorted list?
 (a) Serial search (b) Binary search (c) Linear search (d) None
4. Which of the following sort method is also known as exchange sort?
 (a) Bubble (b) Merge (c) Selection (d) Insertion
5. Complexity of linear search algorithm is:
 (a) $O(n)$ (b) $O(\log n)$ (c) $O(n^2)$ (d) $O(n \log n)$
6. Which of the following sorting algorithm is of divide-and-conquer type?
 (a) Bubble (b) Insertion (c) Merge (d) Selection
7. The complexity of Binary search algorithm is:
 (a) $O(n)$ (b) $O(\log n)$ (c) $O(n^2)$ (d) $O(n \log n)$
8. Which of the following algorithm does not divide the list:
 (a) Linear search (b) Binary search (c) Merge sort (d) Quick sort
9. Which of the following is not a stable sorting algorithm?
 (a) Insertion sort (b) Selection sort (c) Bubble sort (d) Merge sort
10. Which of the following is a stable sorting algorithm?
 (a) Merge sort (b) Typical in-place quick sort
 (c) Heap sort (d) Selection sort
11. Which of the following is not an in-place sorting algorithm?
 (a) Selection sort (b) Heap sort (c) Quick sort (d) Merge sort
12. Running merge sort on an array of size n which is already sorted is:
 (a) $O(n)$ (b) $O(n \log n)$ (c) $O(n^2)$ (d) None
13. The time complexity of a quick sort algorithm which makes use of median, found by an $O(n)$ algorithm, as pivot element is:
 (a) $O(n^2)$ (b) $O(n \log n)$ (c) $O(\log n)$ (d) $O(n)$
14. Which of the following is not a non-comparison sort?
 (a) Counting sort (b) Bucket sort (c) Radix sort (d) Shell sort
15. If the given input array is sorted or nearly sorted, which of the following algorithm gives the best performance?
 (a) Insertion sort (b) Selection sort (c) Quick sort (d) Merge sort
16. Time complexity of bubble sort in best case is:
 (a) $\theta(n)$ (b) $\theta(n \log n)$ (c) $\theta(n^2)$ (d) $\theta(n(\log n)^2)$
17. Counting sort performs Numbers of comparisons between input elements.
 (a) 0 (b) n (c) $n \log n$ (d) n^2
18. Merge sort uses:
 (a) Divide-and-conquer (b) Backtracking
 (c) Heuristic approach (d) Greedy approach

19. What is recurrence for worst case of Quick Sort and what is the time complexity in Worst case?
- (a) Recurrence is $T(n) = T(n-2) + O(n)$ and time complexity is $O(n^2)$
 (b) Recurrence is $T(n) = T(n-1) + O(1)$ and time complexity is $O(n^2)$
 (c) Recurrence is $T(n) = 2T(n/2) + O(n)$ and time complexity is $O(n \log n)$
 (d) Recurrence is $T(n) = T(n/10) + T(9n/10) + O(n)$ and time complexity is $O(n \log n)$
20. Time required to merge two sorted lists of size m and n , is:
- (a) $O(m | n)$ (b) $O(m + n)$ (c) $O(m \log n)$ (d) $O(n \log m)$
21. What is the average case complexity of bubble sort?
- (a) $O(n \log n)$ (b) $O(\log n)$ (c) $O(n)$ (d) $O(n^2)$
22. What is the best case complexity of Quick Sort?
- (a) $O(n \log n)$ (b) $O(\log n)$ (c) $O(n)$ (d) $O(n^2)$

Answers of MCQs

01 (c)	02 (a)	03 (b)	04 (a)	05 (a)	06 (c)	07 (b)	08 (a)	09 (b)	10 (a)
11 (d)	12 (b)	13 (b)	14 (d)	15 (a)	16 (a)	17 (a)	18 (a)	19 (b)	20 (b)
21 (d)	22 (a)								

- Q.2 Describe how binary search works.
- Q.3 Write an algorithm to find the smallest and the largest values in an array.
- Q.4 Write an algorithm to search the position of the largest value in an array.
- Q.5 Describe how selection sort and merge sort work.
- Q.6 Write a program to sort a list of names in descending order using insertion sort.
- Q.7 Differentiate between the following:
- Insertion sort & Bubble sort
 - Selection sort & Merge sort
 - Sequential search & Binary search
- Q.8 Sort the sequence 13, 11, 44, 1, 51, 19, 22, 6, 15 using insertion sort.
- Q.9 Trace the process to sort the following array in ascending order using quick sort algorithm:
- 76, 34, 45, 34
- Q.10 Trace the process to sort the following array in ascending order using insertion sort algorithm:
- 21, 20, 19, 18, 17
- Q.11 What is hashing and different hash functions?
- Q.12 Sort the following data in ascending order using the selection sort algorithm.
- 30, 354, 10, 43, 21, 57, 68, 47, 29
- Q.13 Write an algorithm to perform sequential/linear search on an array.

In previous chapters, we have learned that data can be arranged in memory using different structures. Each structure has its own advantages and disadvantages. For example, we have learned about the array data structure. Arrays have certain disadvantages when used as data storage structures. Some disadvantages are as follows:

- ★ In multiuser environments, a number of users share the same main memory of a computer system. There may not be sufficient adjacent memory space available to hold an array but there could be enough small free blocks of memory that can collectively accommodate the array.
- ★ An array is a static data structure. The size of the array cannot be changed during the program execution (although it can be changed in a vector).
- ★ The data access speed slows down due to the increase in the size of the array.

The linked list solves some of these problems. The linked list is probably the second most commonly used general-purpose storage structure after array. It is a dynamic data structure that uses pointers for its implementation. The linked list is a versatile mechanism suitable for use in many kinds of general-purpose data-storage applications. In fact, we can use a linked list in many cases where the array is used, to improve the performance of applications. Mostly linked lists are used to create trees, graphs, and other abstract data types.

9.1 LINKED LISTS

Linked list is a dynamic linear data structure. It consists of a list or chain of objects (or data elements), each pointing to the next object using a pointer. Each object or data element of the linked list is called a *node*. Each node contains the data and the memory address or location of the next node in the list. Thus each node of the linked list has two parts:

- i) **Data part:** It contains actual data. It can consist of one or more fields. These fields are known as *data*, *information*, *value*, *cargo*, or *payload fields*. A node may contain data of any type. However, all nodes of the list contain the same type of data.
- ii) **Link part:** It contains the memory address of the next node. It is usually known as the *next link* or *next pointer*. It points to the next node (and previous node also for double linked list) in the list. Each node of the list is linked logically with the next node through the *next pointer*. If there is no next node, then a NULL value is stored in it, to mark the end of the list.

A simple linked list is shown in the following figure:

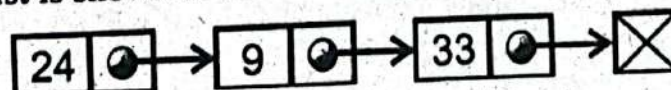


Figure 9.1: Example of a simple linked list

In this figure, each node of the linked list contains two fields: one field contains an integer value and the other contains a link to the next node. The last node is linked to a terminator that indicates the end of the list.

Real Life Example of Linked List

A simple real-life example of a linked list is a train which is a chain of buggies or couches. Each buggy or couch is connected to the next couch. Each couch of the train is like a node. The people's seating arrangement inside the coach represents the data part, while the connection between the two buggies represents the "link part" of the linked list. We can traverse from one couch to another.

Like the linked list, trains also have the last coach which is not further connected to any of the buggies. The engine can be called as the first node of the linked list.

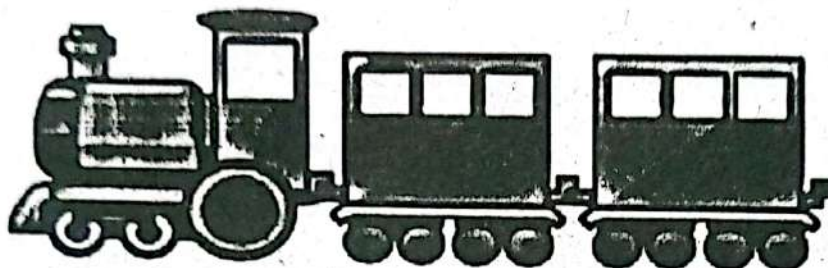


Figure 9.2: Train with buggies

Types of Linked Lists

There are two main types of linked lists: singly linked list and doubly linked list.

9.1.1 Advantages of Linked Lists

Following are some advantages of linked lists as compared to arrays:

- i) **Dynamic Data Structure:** Linked list is a dynamic data structure, which can grow and shrink (by allocating and de-allocating memory) during the execution of the program. So there is no need to define an initial size in the source code for a linked list.

On the other hand, an array is a static data structure that has a fixed size. The size of an array is fixed; therefore, we need to know the upper limit or maximum size beforehand to declare an array. This can lead to a wastage of memory.

- ii) **Insertion and Deletion of nodes:** Insertion and deletion of nodes of the linked list are comparatively easier. There is no need for shifting nodes during the insertion and deletion operations of nodes. When a node is inserted or removed, only the link or pointer field of the next node is updated.

On the other hand, deletion and insertion of elements in the array are difficult. It needs the shifting of elements in an array for insertion and deletion operations.

- iii) **Memory Utilization:** In the case of an array, there is a lot of memory wastage. For example, if we declare an array of size 50 and store only 16 elements in it then space of

34 elements is wasted. There is no such problem in the linked list as memory is allocated only when required. So linked list data structure provides efficient memory utilization.

- (iv) **Implementation of Stack and Queue:** Stack and queue linear data structures can easily be implemented using a linked list.

9.1.2 Disadvantages of Linked Lists

Following are some disadvantages of linked lists as compared to arrays:

- i) **Wastage of Memory:** Linked lists may use more memory than arrays because pointers are also stored along with data elements. Extra memory space for a pointer is required with each data element of the list.
- ii) **No Random Access:** Random access of nodes is not possible in a linked list. We have to access nodes of the linked list sequentially starting from the first node. For example, if we want to access a node at position 'n', then we have to traverse all the nodes before it. However, random access of an element is possible in the case of an array.
- iii) **Reverse Traversing:** In a singly linked list, reverse traversing is difficult (i.e. traversing the singly linked list from the end). It can be traversed only from the beginning. However, a doubly-linked list is somewhat easier to traverse from the end, however it consumes more storage space for the back pointer. In an array, reverse traversing is easier than a linked list.
- iv) **Time Complexity:** Individual nodes are not stored in the contiguous memory locations. More time is required to access individual nodes within the list. In the case of an array, individual elements are stored in the contiguous memory locations. So less time is required to access individual elements of an array.
- v) **Heap Space Restriction:** Whenever memory is dynamically allocated, it utilizes memory from the heap¹. Memory is allocated to the linked list at run time if and only if there is space available in heap. If there is insufficient space in the heap then no memory is allocated.

9.1.3 Pointer & Linked Lists

Pointer is the most important feature of C++ (and C). It is used to create and manipulate dynamic data structures such as linked lists. The concept of a pointer is fundamental to understand the concept of linked lists.

A pointer is a variable that is used to store the memory address of another variable. It is also simply called a *pointer variable*. The data type of the pointer and of the variable whose address it stores must be the same.

¹ Heap is the portion of memory where *dynamically allocated* memory resides. Memory allocated from the heap will remain allocated until one of the following occurs:

- This memory is de-allocated
- The program is terminated

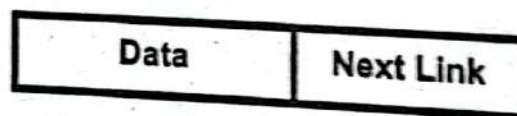
A linked list is accessed via a pointer to the first node or head of the list. The other nodes of the list are accessed via the next-pointer, which is a member of each node. The next-pointer of the last node of a list is set to NULL to mark the end of the list. Usually, data is stored in a linked list dynamically. It means that each node is created during program execution (and thus nodes can also be deleted if required).

The use of pointers to link elements of a data structure implies that the elements need not be physically adjacent in the memory. They can be stored anywhere in the memory but they logically remain adjacent. This type of storage is called *linked storage*.

9.2 SINGLY LINKED LIST

A type of linked list in which each node contains data and the address of the next node is called a *singly linked list*. Sometimes, it is also simply called a *single linked list*. It is named a *singly linked list* because its every node contains a single link (address) of the next node and does not contain the address of the previous node. It can be accessed or visited from beginning to its end i.e. only in one direction. The singly linked list is also known as a *one-way list* or *one-way chain*.

In a singly linked list, each node consists of at least two fields. The first field contains the data. This field is called the *data field* (multiple data fields can be used in a node to store information). The second field contains the pointer or link (memory address) to the next node in the list. This field is called the *next link* or *next pointer field*. The structure of a node of a singly linked list is shown in the following figure;



The *next link field* of the last node contains a NULL value. The singly linked list that has no node is called an *empty list*. It has a value NULL. Following figure shows a singly linked list with two data fields and one next link field.

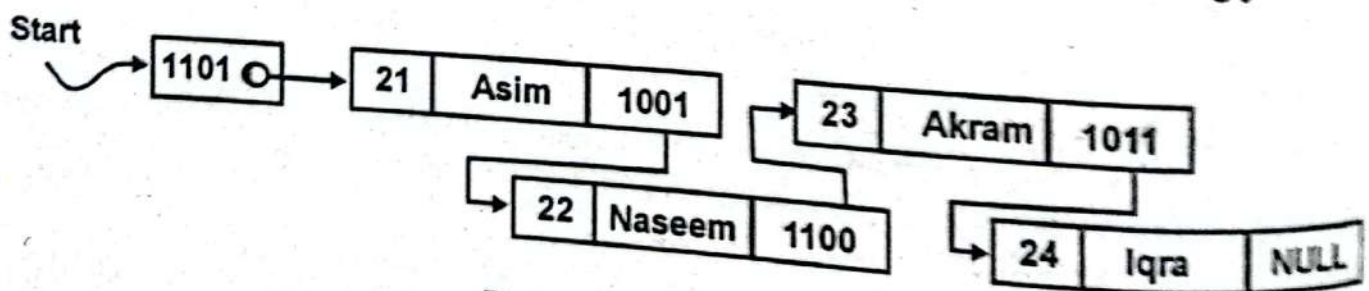


Figure 9.3: Singly Linked List

In a singly linked list, the address of the first node is always stored in a reference node known as the *front*, *head*, or *start*. It is the entry-point into the linked list. A linked list is accessed with the reference to the address of its first node i.e. *head* of the linked list. For example, if we want to access a particular node of the list, then we have to start from the *head* or *Start*. The value of *Start* is NULL for an empty list.

In figure 9.3 of the singly linked list, "Start" is used as the *head* of the list. It contains the memory address of the first node of the list. Suppose the memory address of the first node is 1101. The second node is created and its memory address is stored in the next pointer field of the first node. Suppose its memory address is 1001. In the above figure, 1100 and 1011 represent the memory addresses of the third node and fourth node respectively. The last node of the list contains a NULL value in its next-pointer field.



Please note that head or start is not a separate node, it is only the reference to the first node in the linked list. It is a pointer variable of type 'node'.

9.2.1 Implementation of Singly Linked List

The storage technique of the linked list is different from the linear array. The linked list uses dynamic memory allocation i.e. it allocates memory for new list nodes as needed. The nodes of a linked list are stored in memory in scattered form but these are linked together through the next link field of the node. In C++, a linked list can be implemented using structure and pointers.

Declaring Structure of a Node

Before creating a new node, its structure is declared/defined. Following is the syntax to declare the structure of a node of a linked list in C++:

```
struct node
{
    type_1 data_1;
    type_2 data_2;
    -----
    -----
    type_n data_n;
    node *link;
};
```

In this syntax, data_1, data_2, ..., data_n represent the data fields of data types type_1, type_2, ..., type_n respectively. Similarly, the "link" is a pointer variable. Please note that in place of a data type, "node" is written before *link. That's because it is a *self-referencing pointer*. It means a pointer that points to whatever it is a part of. Here "link" is a part of a node and it will point to the next node.

For example, a structure named "student_node" is defined as follows:

```
struct student_node
{
    int roll_no;
    char name[15];
    student_node *link;
};
```


In this example, two data fields "roll_no" and "name" are defined to store data values in the node. The "link" is defined as a pointer to the object "student_node", which will be used as the next-pointer. It is used to store the address of the next node.

Creating a Node

In C++, the 'new' operator is used to create a new node during program execution. This operator allocates memory for the specified object and returns its memory address to the pointer variable. Its general syntax is as follows;

```
ptr = new object_name;
```

In this syntax:

ptr It represents the pointer to an object.

object_name It represents the name of the object. It may be the name of a defined class or structure.

Following C++ statements declare a pointer variable "start" to an object "student_node" and create a node of 'student_node':

```
student_node *start;
```

```
start = new student_node;
```

The first statement declares the pointer variable 'start' of type "student_node". It is used to store the memory address of the first or starting node of the singly linked list. In the second statement, the 'new' operator is used. It creates a new node and assigns its memory address to the pointer variable 'start'.

Accessing Data of a Node

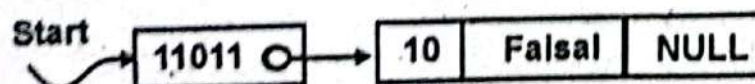
The operator "->" (hyphen and greater than sign) is used to access the data of the pointer type object (node of a linked list). In order to store data values in fields 'roll_no', 'name' and 'NULL' value to link pointer of the node 'start', the assignment statements are as follows:

```
start->roll_no = 10;
```

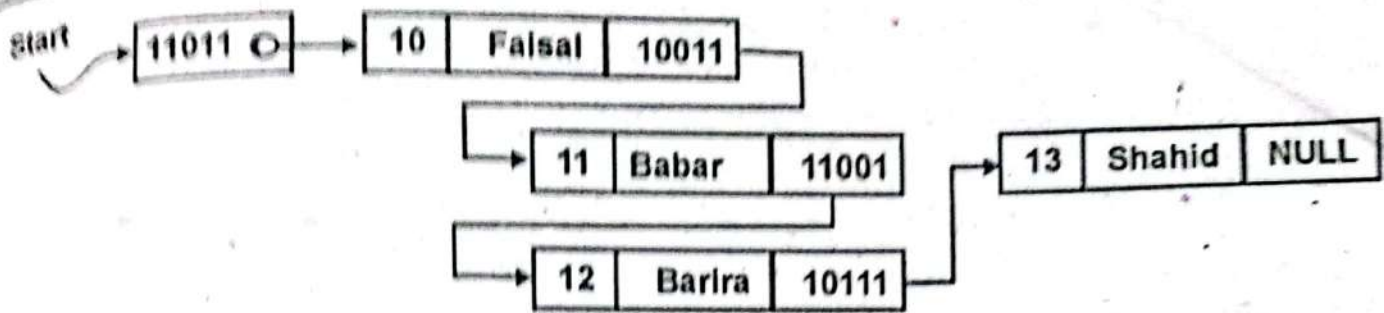
```
start->name = "Faisal";
```

```
start->link = NULL;
```

Suppose the memory address 11011 is assigned to the pointer variable "start". The storage in memory will be as shown in the following figure:



While creating another new node, the address of the previous node is stored in a pointer variable. The memory address of the newly created item is stored in the pointer field of the previous node. The representation of a singly linked list having nodes in memory is shown in the following figure:



The above list is a linear linked list. It does not occupy consecutive memory locations (like an array). Each element or item of the list is at a different location. These are at memory addresses 11011, 10011, 11001, and 10111. Each node has a pointer that holds the address of the next node. The pointer of the last node does not point to any node. It contains a NULL value. The whole list is accessed with reference to the pointer variable that holds the starting address of the linked list.

Deleting a Node

The memory allocated through the 'new' operator will be reserved for a node until the related program is terminated. This may create many problems if the size of the program is very large.

We can de-allocate the memory allocated with the 'new' operator for a node, and it can be re-allocated for other parts of the current program or operating system. The 'delete' operator is used to de-allocate memory occupied by a node. The general syntax of the 'delete' operator is as follows;

```
delete ptr;
```

In this syntax, "ptr" represents the pointer variable which holds the memory address of the node which is to be deleted from memory.

Algorithm – Creating Linked List

Write an algorithm that creates a linked list of values consisting of ten nodes and displays the data of nodes on the screen.

1. START
2. Define the structure of the node. Suppose the structure of the node is as follows;


```

      struct node
      {
          int data;
          node *link;
      };
      
```
3. Declare three pointer variables START, TEMP, and CURRENT of type node
4. REPEAT step-5 to 10 FOR C = 1 to 10

5. INPUT data value for the node in X
6. Create a TEMP node in memory
7. Assign the values in TEMP node
 TEMP->data = X
 TEMP->link = NULL
8. [If the list is empty, then assign the address of TEMP to START]
 IF START = NULL then START = TEMP
9. [If the list is not empty then go to the end of the list. First, assign the value of START into CURRENT, and then go to the end of the list via CURRENT pointer]
 CURRENT = START
 WHILE(CURRENT->link != NULL) [go to end of list]
 CURRENT = CURRENT->link
 END WHILE
10. [Assign value of the newly created node to the link field of the last node]
 CURRENT->link = TEMP
11. Display the data of all nodes of the list
12. EXIT

Program Code 9.1

// program that creates a new linked list of student records and
 // displays the records on screen.

```
#include <iostream.h>
#include <string.h>
#include <conio.h>
```

```
struct std_node
{
    int roll_no;
    char name[15];
    std_node *link;
};
```

```
class std_list
```

```
{
    private:
        std_node *start, *current, *temp;
    public:
        std_list(void)
        {
            start = NULL;
        }
}
```

```
void insert(int, char []);  
void display(void);
```

```
};
```

```
void main(void)
```

```
{
```

```
    std_list obj;
```

```
    int rn, rec = 1;;
```

```
    char nm[15];
```

```
    clrscr();
```

```
    cout<<"Enter records of five students"<<endl;
```

```
    do
```

```
    {
```

```
        cout<<"Enter Record # "<<rec<<endl;
```

```
        cout<<"Enter roll number : ";
```

```
        cin>>rn;
```

```
        cout<<"Enter name : ";
```

```
        cin>>nm;
```

```
        // add record into list by calling function 'insert()'
```

```
        obj.insert(rn, nm);
```

```
        rec++;
```

```
    } while(rec<=5);
```

```
    // display records of students stored in linked list
```

```
    cout<<endl<<"Data of the list: "<<endl;
```

```
    obj.display();
```

```
    getch();
```

```
} //end of main()
```

```
    // member function to add new node
```

```
void std_list::insert(int rn, char nm[])
```

```
{
```

```
    // create a temp node in memory
```

```
    temp = new std_node;
```

```
    // assign data to node temp
```

```
    temp->roll_no = rn;
```

```
    strcpy(temp->name, nm);
```

```
    temp->link = NULL;
```

```
    // if linked list is empty then assign address of temp to start
```

```
    if(start == NULL)
```

```
    {
```

```
        start = temp;
```

```
    }
```


9.2.2.1 Traversing a Singly Linked List

Traversing a linked list is important for many applications. It is a process of going through all the nodes of the linked list. Singly linked list is visited from the beginning to the end of the list i.e. until NULL is found in the link or pointer field of the node. A linked list is usually visited to display the data of its nodes or to count the total number of nodes of the list.

Algorithm – Traversing Singly Linked List

Write an algorithm that displays data of all the nodes of a singly linked list. Suppose "START" stores the starting address of the list and each node of the list has two fields. The first is "data" field for storing data and the other is "link" field for storing address of the next node.

1. START
2. Declare a pointer variable CURRENT of type node
3. IF linked list is empty then display a message and exit.
4. [Assign the starting address of the list to CURRENT pointer variable]
CURRENT = START
5. [Access data of list via CURRENT pointer variable]
REPEAT step-6 to 7 WHILE CURRENT!= NULL
6. PRINT CURRENT->Data
7. CURRENT = CURRENT->Link [Move to next node]
[End of step-5 loop]
8. EXIT

display() function

// this function displays data of linked list

void display(void)

```
{  
    struct node *current;  
    if(start==NULL)  
    {  
        cout<<"List is empty"<<endl;  
        return;  
    }  
    else  
    {  
        current = start;  
        while(current!=NULL)
```

```

    {
        cout<<current->data<<endl;
        current = current->link;
    }
}
} //end of display()

```

Algorithm – Counting Nodes

Write an algorithm that counts total number of nodes in a singly linked list. Suppose "START" stores the starting address of the list.

1. START
2. Declare a pointer variable CURRENT of type node
3. IF linked list is empty then display a message and exit.
4. [Assign the address of starting address of list to CURRENT pointer variable]
CURRENT = START
5. COUNT = 0
6. [Access data of list via CURRENT pointer variable]
REPEAT Step-7 to 8 WHILE CURRENT!= NULL
7. COUNT = COUNT + 1
8. CURRENT = CURRENT->Link
[End of step-6 loop]
9. PRINT "Total nodes =", COUNT
10. EXIT

count() function

// this function counts the total number of nodes in a linked list and
// returns the result to the calling function.

```

int count(void)
{
    struct node *current;
    int n = 0;
    if(start==NULL)
    {
        cout<<"List is empty"<<endl;
        return 0;
    }
    else
    {
        current = start;

```



```

while(current!= NULL)
{
    n++;
    current = current->link;
}
return n;
}
} //end of count()

```

Complexity Analysis of Traversal Algorithm of Singly Linked List

A linked list can only be accessed via its head node. The traversal algorithm of singly linked list uses only one loop for traversing list of 'n' nodes. All nodes of the list have to be accessed in the worst-case (from head node to last node one by one). Therefore, time complexity and space complexity of singly linked list traversal algorithm are as follows:

Time Complexity		Space Complexity
Average-Case	Worst-Case	Worst-Case
$\Theta(n)$	$O(n)$	$O(n)$

9.2.2.2 Searching in Singly Linked List

In searching operation, a particular data item is searched from a linked list. In this operation, linked list is visited sequentially from beginning up to the required data item. The search is "successful" if the specified data item is found in the list. Search operation is terminated as soon as a data item is found. If the specified data item is not found in the list, the search is declared as "unsuccessful."

Searching operation is frequently used along with insertion, deletion and update operations. For example, when a data item is to be deleted, it is first searched and then deleted, if found.

Algorithm – Searching in Singly Linked List

Write an algorithm that searches a specific data item from a singly linked list. Suppose pointer variable "START" stores the starting address of the list.

1. START
2. IF linked list is empty then display a message and exit.
3. flag = 0
4. Declare a pointer variable CURRENT of type node
5. INPUT value to search in X

6. [Assign the starting address of the list to CURRENT pointer variable]

CURRENT = START

7. [Search the data item from the list via CURRENT pointer variable]

WHILE CURRENT != NULL

IF CURRENT->Data = X THEN

flag = 1

EXIT

END IF

END WHILE

8. IF flag = 1 THEN

PRINT "Value found in the list"

ELSE

PRINT "Value not found in the list"

END IF

9.

EXIT

search() function

// this function searches a particular value from a linked list

void search (void)

{

struct node *current;

int value, flag = 0;

if(start == NULL)

{

cout << "List is empty" << endl;

return;

}

else

{

cout << "Enter a value to search :";

cin >> value;

current = start;

while(current != NULL)

{

if(current->data == value)

{

flag = 1;

break;

}

current = current->link;

}




```

    }
    if(flag ==1)
        cout<<"Value found in the list";
    else
        cout<<"Value not found in the list";
} //end of search()

```

Complexity Analysis of Search Algorithm of Singly Linked List

The search algorithm of singly linked list is similar to Traversal algorithm. It also uses only one loop for searching list of 'n' nodes or items. When the required node is in the last position of the list or does not exist in the list, then the algorithm moves from first node up to the last node. All the nodes of the list have to be accessed in the worst-case. Therefore, time complexity and space complexity of singly linked list search algorithm are as follows:

Time Complexity		Space Complexity
Average-Case	Worst-Case	Worst-Case
$\Theta(n)$	$O(n)$	$O(n)$

9.2.2.3 Insertion in a Singly Linked List

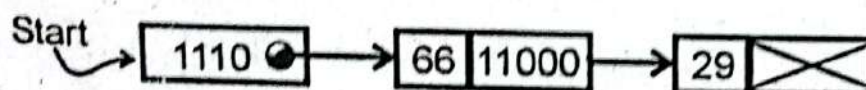
The insertion operation is used to insert a new node in the linked list. A new node can be inserted in a linked list at the end of list, beginning of list, or at any specified location in the list. The new node is inserted in the list by simply interchanging / updating the pointer or link fields.

i) Insertion of a new node at the end of the list

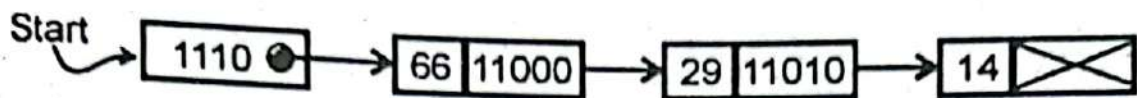
Insertion of a new node at the end of the list is very easy. We need to traverse the list up to the last node. We create a new node and assign values to it (in link field of the node, we always assign NULL value). Then we assign the memory address of the newly created node to the link field of the last node of the existing list. The newly created node will be added (or inserted) at the end of the list and becomes the last node of the list.

In case of empty list, we simply assign the memory address of the newly created node to the start or head variable (pointer variable) that points the starting address of the list. The newly created node becomes the first node of the list.

Suppose a linked list having two nodes is shown in the following figure:



Suppose the memory address of the newly created node is 11010 and a value 14 is assigned to its data field. The list after insertion of new node at the end is shown in the following figure:



Algorithm – Insertion of a new node at the end of a linked list

Write an algorithm that inserts a new node at the end of a linked list. Suppose pointer variable "START" stores the starting address of the list.

1. START
2. Declare two pointer variables TEMP and CURRENT of type node
3. Create a linked list.
4. Create a new node TEMP.
5. INPUT data value in variable X for the new node to be inserted
6. [Assign values to the TEMP node]
 - TEMP->data = X
 - TEMP->link = NULL
7. [Go to the end of the linked list]
 - CURRENT = START
 - WHILE CURRENT->link != NULL
 - CURRENT = CURRENT->link
 - END WHILE
8. [Assign the address of the newly created node TEMP to the link field of the last node.]
 - CURRENT->link = TEMP
9. EXIT

Insert_end() function

// this function inserts a new node at the end of a linked list

void insert_end (void)

```

{
    struct node *temp, *current;

    // create a temp node in memory
    temp = new node;

    // input or assign data to temp node
    cout<<"Enter the data value for the node:";
    cin>>temp->data;
  
```



```

temp->link = NULL;
//If the list is empty
if(start==NULL)
{
    start = temp;
}
else
{
    // go to end of the list
    current = start;
    while(current->link != NULL)
        current = current->link;

    // assign the address of temp to the link field of last node
    current->link = temp;
}
} //end of insert_end()

```

ii) Insertion of a new node at the beginning of the list

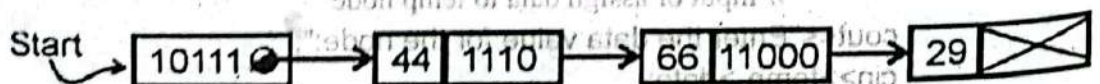
Insertion of a new node at the beginning of the list is very simple process; because there is no need to visit the list (go to the end of list). We create a new node and assign data value(s) to it (in link field of the node, NULL value is not assigned). After this step, we assign the starting address of the existing list to the link field of the newly created node and then assign the memory address of the newly created node to the 'start' or 'head' pointer variable that points the starting address of the list. Thus, the newly created node becomes the first node of the list.

In case of empty list, we simply assign the memory address of the newly created node to the start or head variable (pointer variable) that points to the starting address of the list. Thus, the newly created node becomes the first node of the list.

Suppose a linked list having two nodes is shown in the following figure:



Further assume that the memory address of the newly created node is 10111 and a value 44 is assigned to its data field. The starting address of the list "1110" will be assigned to link field of the newly created node and address 10111 of the newly created node will be assigned as a starting address of the list.



Algorithm – Insertion at the beginning of a linked list

Write an algorithm that adds a new node at the beginning of a linked list. Suppose pointer variable "START" stores the starting address of the list.

1. START
2. Declare a pointer variable TEMP of type node.
3. Create a linked list.
4. Create a new node TEMP
5. INPUT data value for TEMP in X
6. [Assign value to the data field of TEMP node.]
TEMP->data = X
7. [Assign the value of START (starting address of the list) to the link field of the newly created node and then assign address of TEMP to START.]
TEMP->link = START
START = TEMP

8. EXIT

Insert_begin() function

// this function inserts a node at the beginning of a linked list

void insert_begin(void)

{
struct node *temp;

// create a temp node in memory

temp = new node;

// input or assign data to temp node

cout<<"Enter the data value for the node:";

cin>>temp->data;

temp->link = NULL;

// if the list is empty

if(start==NULL)

start = temp;

{

else

// assign the starting address of the list to the link field of temp and
// address of temp to the 'start' variable

temp->link = start;

start = temp;

}

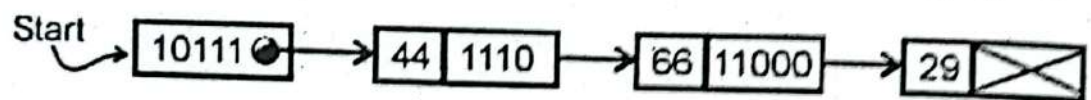
} //end of insert_begin()

iii) Insertion of a new node at a specific location of a list

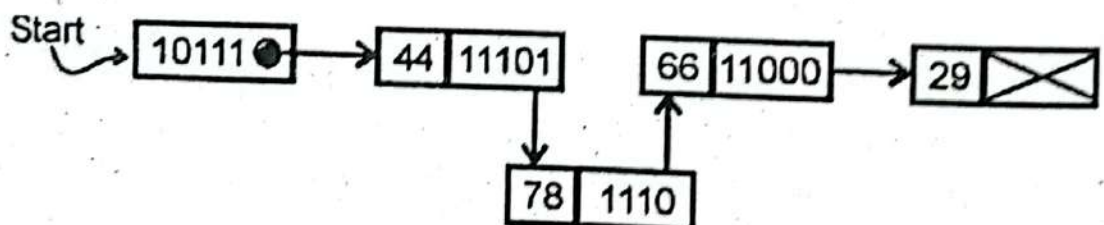
Sometimes we have to insert a new node at a specific location of a list. In this case, we traverse the list to find the place to insert the node. In the traversal process, we must maintain a reference to previous and next nodes of the list. Then we will insert a new node between previous and next. We create a node and assign values to its data field(s). We assign the value of link field of current node to the link field of the newly created node. Then we assign the memory address of the newly created node to the link field of current node.

In case of empty list, we simply assign the memory address of the newly created node to the start or head (pointer variable) that points the starting address of the list. The newly created node becomes the first node of the list.

Suppose a linked list having three nodes is shown in the following figure:



Suppose the memory address of the newly created node is 11101 and a value 78 is assigned to its data field. Suppose we want to insert a node after first node of the list. The address 1110 stored in link field of first node will be assigned to link field of the newly created node and address of the newly created node i.e. 11101 will be assigned to the link field of first node. After inserting the new node, the list will be as shown in the following figure:



Algorithm – Insertion at a specified location within a Linked List

Write an algorithm that adds a new node at a specified location within a linked list. Suppose pointer variable "START" stores the starting address of the list.

1. START
2. Declare two pointer variables TEMP and CURRENT of type node
3. Create a linked list with three nodes. Suppose pointer variable "START" stores the starting address of this list.
4. Create a new node TEMP
5. INPUT data value for TEMP in X
6. [Assign the value of X to the data field of TEMP node.]
TEMP->data = X

7. INPUT a value in Y to be searched in the list after which the new node is to be inserted.
8. [Assign the value of starting address of linked list into pointer variable CURRENT.]
CURRENT = START
9. [Go to the node that has value equal to Y via CURRENT pointer (i.e. after which the new node is to be inserted).]
WHILE CURRENT!=NULL
IF CURRENT->data = Y THEN
EXIT
END IF
CURRENT = CURRENT->link
END WHILE
10. [Assign the value of link field of the current node to the link field of TEMP (newly created node). Similarly, assign the address of the newly created node (i.e. address of TEMP) to the link field of the current node.]
TEMP->link = CURRENT->link
CURRENT->link = TEMP
11. EXIT

Insert_location() function

// this function inserts a new node at a particular location in the linked list

void insert_location(void)

```
{
    struct node *temp, *current;
    int n, pos = 0;

    // if the list is empty
    if(start==NULL)
    {
        cout<<"List is empty";
        return;
    }
    else
    {
        cout<<"Enter the value to search:";
        cin>>n;

        // search the value 'n' in the list
        current = start;
        while(current!= NULL)
        {
            // if value found
            if(current->data == n)
            {
```



```

        pos = 1;
        break;
    }
    current = current->link;

    // if value found then insert the node at that position
    if(pos == 1)
    {
        // create a temp node in memory
        temp = new node;
        // input data to the newly created node 'temp'
        cout<<"Enter the data value for the node:";
        cin>>temp->data;
        // interchange the link values of temp and current
        temp->link = current->link;
        current->link = temp;
    }
    else
    {
        cout<<"Value not found";
        return;
    }
} //end of insert_location()

```

Complexity Analysis of Insertion Algorithm of Singly Linked List

The insertion algorithm of singly linked list moves from the head node of list, before the node where new node is to be inserted. Then it inserts a new node into the list. Thus insertion algorithm of singly linked list takes $O(n)$ time for inserting new node, where 'n' is the number of nodes in the list. Therefore, time complexity and space complexity of insertion algorithm of singly linked list are as follows:

Time Complexity		Space Complexity
Average-Case	Worst-Case	Worst-Case
$\Theta(n)$	$O(n)$	$O(n)$

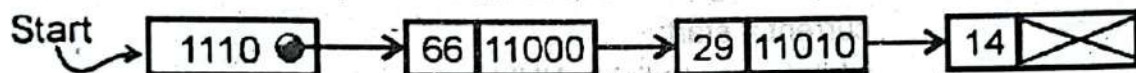
9.2.2.4 Deletion in Singly Linked List

In deletion operation, a node is deleted from the list. Deletion operation in singly linked list is very simple and straightforward. A node can be deleted from the end of list, beginning of list, or at any specified location in the list.

i) Deletion of a node from the end of a list

Deletion of a node from the end of the list is very simple. Delete the last node of the list and assign the link field of the second last node, with NULL value. To perform these actions, read the list from the beginning to the end. In each step, assign the memory address of each node to a pointer variable (of type node) such as PREVIOUS. When end of the list is reached (i.e. value of link field becomes NULL), the PREVIOUS pointer will hold the address of the last node of the list. This address is stored in the link field of the second last node of the list. Delete the last node and assign NULL value to the link field of the second last node of the list.

Suppose a linked list having three nodes as shown in the following figure:



After deleting the last node, the linked list will become as shown below:



Algorithm – Deleting a node from the end of a list

Write an algorithm that deletes a node from the end of a linked list. Suppose pointer variable "START" stores the starting address of the list.

1. START
2. Declare two pointer variables PREVIOUS and CURRENT of type node.
3. Create a linked list.
4. [Go to the end of the linked list.]
 CURRENT = START
 WHILE CURRENT->link != NULL
 PREVIOUS = CURRENT
 CURRENT = CURRENT->link
 END WHILE
5. [Delete the CURRENT (last node) and assign NULL value in the link field of PREVIOUS.]
 delete CURRENT
 PREVIOUS->link = NULL
6. EXIT

delete_end() function

// this function deletes a node from the end of a linked list

void delete_end (void)


```

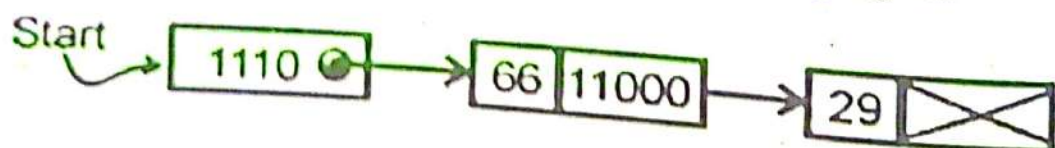
struct node *previous, *current;

    if the list is empty
    if(start==NULL)
    {
        cout<<"List is empty";
        return;
    }
    else
    {
        go to end of the list
        current = start;
        while(current->link!= NULL)
        {
            previous = current;
            current = current->link;
        }
        cout<<"Data of last node is : "<<current->data;
        previous->link = NULL;
        delete current;
        cout<<endl<<"Last node is deleted";
    }
} // end of delete_end()

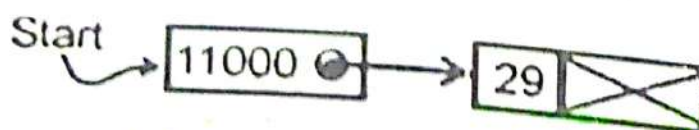
```

ii) Deletion of a node from the beginning

To delete first node of the list, simply assign the value of link field of first node of the list to the pointer variable START, which holds the starting address of the list. Suppose a linked list having two nodes as shown in the following figure:



After deleting the first node, the linked list will become as shown below:



Algorithm – Deleting a node from the beginning of a List

Write an algorithm that deletes a node from the beginning of a linked list. Suppose pointer variable "START" stores the starting address of the list.

1. START
2. Declare a pointer variable CURRENT of type node
3. Create a linked list.
4. [Assign the starting address of the list to CURRENT pointer variable and then assign the value of link field of CURRENT (i.e. first node) to START.]
 CURRENT = START
 START = CURRENT->link
5. [Delete the CURRENT.]
 Delete CURRENT
6. EXIT

delete_begin() function

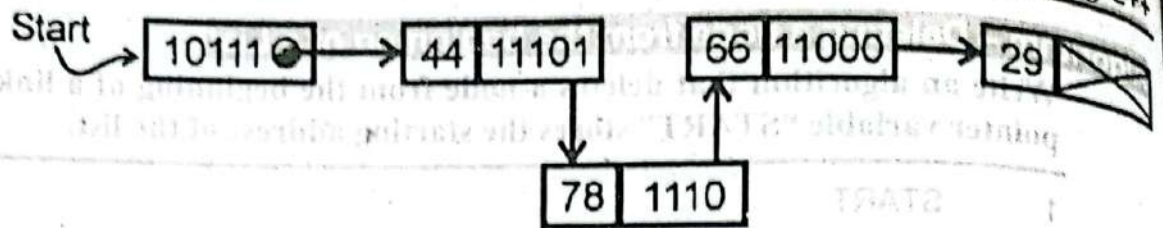
// this function deletes the first node of a linked list

```
void delete_begin(void)
{
    struct node *current;
    if(start==NULL)
    {
        cout<<"List is empty";
        return;
    }
    else
    {
        current = start;
        start = current->link;
        cout<<"Data of first node : "<<current->data;
        delete current;
        cout<<endl<<"First node is deleted";
    }
}
//end of delete_begin()
```

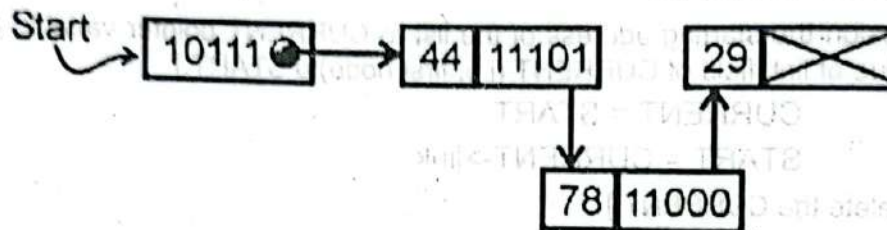
iii) Deletion of a node at a particular Location

Deleting a node at a particular location is same as deleting a node from the end of a list. In deleting a node from the end of list, preceding node is assigned NULL value in its link field, while in deleting a node from a particular location, its link field value is assigned to the link field of its preceding node (previous node).

Suppose a linked list having four nodes as shown in the following figure. Suppose we want to delete the third node of the list that has value 66 in its data field.



After deleting the third node, the linked list will become as shown below:



Algorithm – Deletion of a node from a particular location within a Linked List

Write an algorithm that deletes a node from a particular location within a linked list.

1. START
2. Declare three pointer variables START, PREVIOUS and CURRENT of type node.
3. Create a linked list.
4. INPUT data value of a node in-Y that is to be deleted.
5. [Assign the value of starting address of linked list into pointer variables CURRENT and PREVIOUS.]
 $CURRENT = PREVIOUS = START$
6. [Go to the node that has value equal to Y via CURRENT pointer (i.e. node that is to be deleted).]
 $POS = 0$
 WHILE $CURRENT \neq NULL$
 IF $CURRENT \rightarrow data = Y$ THEN
 $POS = 1$
 EXIT
 END IF
 $PREVIOUS = CURRENT$
 $CURRENT = CURRENT \rightarrow link$
 END WHILE
7. IF $POS = 1$ THEN
 [Assign the value of link field of CURRENT node to the link field of PREVIOUS node and then delete the CURRENT node]
 $PREVIOUS \rightarrow link = CURRENT \rightarrow link$
 delete CURRENT
 ELSE
 PRINT "Value not found"
 END IF
8. EXIT

delete_location () function

// this function deletes a node from a particular location in a linked list

void delete_location (void)

struct node *previous, *current;
int pos = 0;

Worst-Case	Average-Case	Best-Case
$O(n)$	$O(n)$	$O(1)$

```
if(start==NULL)
{
    cout<<"List is empty";
    return;
}
```

else

```
int n;
cout<<"Enter a value to search:";
cin>>n;
```

// search the value in the list

```
current = previous = start;
while(current!= NULL)
```

```
{
    if(current->data == n)
```

```
    pos = 1;
    break;
}
```

```
previous = current;
current = current->link;
```

// delete the node if value found in the list

```
if(pos == 1)
```

```
previous->link = current->link;
delete current;
cout<<"Node is deleted";
```

```
}
```

```
else
    cout<<"Value not found : " <<endl;
```

```
} //end of delete_location()
```


Complexity Analysis of Deletion Algorithm of Singly Linked List

Like insertion algorithm, the deletion algorithm of singly linked list moves from the head node of list to the required node to be deleted. Then it deletes the required node from the list. Thus deletion algorithm of singly linked list takes $O(n)$ time for deleting new node. Therefore, time complexity and space complexity of deletion algorithm of singly linked list are as follows:

Time Complexity		Space Complexity
Average-Case	Worst-Case	Worst-Case
$\Theta(n)$	$O(n)$	$O(n)$

9.2.2.5 Sorting in Singly Linked List

Like arrays, sorting operation is also applicable on singly linked list. We can use different methods for sorting data items of a linked list such as bubble sort, merge sort, shell sort, bucket sort, selection sort, radix sort etc. However, merge sort is preferred for sorting linked list.

Algorithm – Sorting Linked List

Write an algorithm that sorts data values of a singly linked list using any sort method.

1. START
2. Create a linked list of unsorted numbers.
3. Sort the numbers of linked list using a suitable sort method.
4. Display the sorted list
5. EXIT

bubble_sort () function

// this function sorts a singly linked list using bubble method

```
void bubble_sort(struct node *start)
```

```
{  
    if(start==NULL)  
    {  
        cout<<"List is empty";  
        return;  
    }  
    struct node *current, *previous;  
    int size, temp;  
    size = count();  
    // find the size of list
```

```
    // sort the linked list  
    for(int u = size; u>1; u--)
```

```
{
```

```

current = start;
for(int i = 1; i < u; i++)
{
    previous = current;
    current = current->link;
    if(previous->data > current->data)
    {
        temp = previous->data;
        previous->data = current->data;
        current->data = temp;
    }
} //end of inner for loop
} //end of upper for loop
} //end of bubble_sort()

```

Program Code 9.2

// Program that implements the operations of singly linked list

```

#include <iostream.h>
#include <conio.h>

```

```

struct node
{
    float data;
    node *link;
};

class list
{
private:
    node *start;
public:
    list(void)
    {
        start = NULL;
    }
    void insert_begin(void);
    void insert_end(void);
    void insert_location(void);
    void delete_begin(void);
    void delete_end(void);
    void delete_location(void);
    void search(void);
    void display(void);
    int count(void);
    void bubble_sort(void);
};

```




```
void main(void)
```

```
{
```

```
    list obj;
```

```
    int op, loop = 1;
```

```
    while(loop)
```

```
    {
```

```
        clrscr();
```

```
        cout<<"01- Insert a node at the beginning"<<endl;
```

```
        cout<<"02- Insert a node at the end"<<endl;
```

```
        cout<<"03- Insert a node at a specific location"<<endl;
```

```
        cout<<"04- Delete a node at the beginning"<<endl;
```

```
        cout<<"05- Delete a node at the end"<<endl;
```

```
        cout<<"06- Delete a node at a specific location"<<endl;
```

```
        cout<<"07- Sort the values of the list"<<endl;
```

```
        cout<<"08- Search the value from a list"<<endl;
```

```
        cout<<"09- Display data of nodes"<<endl;
```

```
        cout<<"10- Count nodes of the list"<<endl;
```

```
        cout<<"11- Exit"<<endl;
```

```
        cout<<"Enter your option [1-11]";
```

```
        cin>>op;
```

```
        switch(op)
```

```
        {
```

```
            case 1: obj.insert_begin(); break;
```

```
            case 2: obj.insert_end(); break;
```

```
            case 3: obj.insert_location(); break;
```

```
            case 4: obj.delete_begin(); break;
```

```
            case 5: obj.delete_end(); break;
```

```
            case 6: obj.delete_location(); break;
```

```
            case 7: obj.bubble_sort(); break;
```

```
            case 8: obj.search(); break;
```

```
            case 9: obj.display(); break;
```

```
            case 10: cout<<"Total nodes are: "
```

```
                      <<obj.count();
```

```
                      getch(); break;
```

```
            case 11: loop = -1; break;
```

```
            default: cout<<"Invalid option"; break;
```

```
        } //end of switch
```

```
        if(loop == -1) break;
```

```
    } //end of while loop
```

```
} //end of main ()
```

```
// this function displays data of the linked list
```

```
void list::display(void)
```

```
{
```

```
    struct node *current;
```

```
    if(start==NULL)
```

```
    {
```

```
        cout<<"List is empty"<<endl;
```

```
        getch();
```

```
        return;
```

```

    }
    else
    {
        current = start;
        while(current!=NULL)
        {
            cout<<current->data<<endl;
            current = current->link;
        }
        getch();
    }
} //end of display()
// this function counts the total number of nodes in the linked list and
// returns the result to the calling function.
int list::count(void)
{
    struct node *current;
    int n = 0;
    if(start==NULL)
    {
        cout<<"List is empty"<<endl;
        getch();
        return 0;
    }
    else
    {
        current = start;
        while(current!= NULL)
        {
            n++;
            current = current->link;
        }
        return n;
    }
} //end of count()

// this function searches a particular value from the linked list
void list::search(void)
{
    struct node *current;
    int value, flag = 0;
    if(start==NULL)
    {
        cout<<"List is empty"<<endl;
        getch();
        return;
    }
    else
    {
        cout<<"Enter a value to search :";
        cin>>value;
    }
}

```



```

        current = start;
        while(current!= NULL)
        {
            if(current->data == value)
            {
                flag = 1;
                break;
            }
            current = current->link;
        }
    }
    if(flag == 1)
        cout<<"Value found in the list";
    else
        cout<<"Value not found in the list";
    getch();
} //end of search()

// this function inserts a new node at the end of the linked list
void list::insert_end (void)
{
    struct node *temp, *current;

    // create a temp node in memory
    temp = new node; // malloc

    // input or assign data to temp node
    cout<<"Enter the data value for the node:";
    cin>>temp->data;
    temp->link = NULL;

    // if the list is empty
    if(start==NULL)
    {
        start = temp;
    }
    else
    {
        // go to end of the list
        current = start;
        while(current->link!= NULL)
            current = current->link;

        // assign the address of temp to the link field of the last node
        current->link = temp;
    }
} //end of insert_end()

// this function inserts a node at the beginning of the linked list
void list::insert_begin(void)

```

```

{
    struct node *temp;

    // create a temp node in memory
    temp = new node;

    // input or assign data to temp node
    cout<<"Enter the data value for the node:";
    cin>>temp->data;
    temp->link = NULL;

    // if the list is empty
    if(start==NULL)
    {
        start = temp;
    }
    else
    {
        // assign the starting address of list to the linked field of temp and
        // address of temp to the start
        temp->link = start;
        start = temp;
    }
} //end of insert_begin()

```

// this function inserts a node at a particular location in the linked list

```

void list::insert_location(void)
{

```

```

    struct node *temp, *current;
    int n, pos = 0;

```

```

    // if the list is empty
    if(start==NULL)
    {
        cout<<"List is empty";
        getch();
        return;
    }

```

```

    else
    {

```

```

        cout<<"Enter the value to search:";
        cin>>n;

```

```

        // search the value 'n' in the list
        current = start;
        while(current!= NULL)
        {

```

```

            // if value found
            if(current->data == n)
            {

```



```

        pos = 1;
        break;
    }
    current = current->link;
}
}
// if value found then insert the node at that position
if(pos == 1)
{
    // create a temp node in memory
    temp = new node;

    // input data to the newly created node 'temp'
    cout<<"Enter the data value for the node:";
    cin>>temp->data;

    // interchange the link value of temp and current
    temp->link = current->link;
    current->link = temp;
}
else
{
    cout<<"Value not found";
    getch();
    return;
}
} //end of insert_location()

// this function deletes a node from the end of the linked list
void list::delete_end (void)
{
    struct node *previous, *current;
    // if the list is empty
    if(start==NULL)
    {
        cout<<"List is empty";
        getch();
        return;
    }
    else
    {
        // go to end of the list
        current = start;
        while(current->link!= NULL)
        {
            previous = current;
            current = current->link;
        }

        cout<<"Data of last node is : "<<current->data;
        previous->link = NULL;
        delete current;
    }
}

```

```

        cout<<endl<<"Last node is deleted";
        getch();
    }
} //end of delete_end()
// this function deletes the first node of the linked list
void list::delete_begin(void)
{
    struct node *current;

    if(start==NULL)
    {
        cout<<"List is empty";
        getch();
        return;
    }
    else
    {
        current = start;
        start = current->link;
        cout<<"Data of first node : "<<current->data;
        delete current;
        cout<<endl<<"First node is deleted";
        getch();
    }
} //end of delete_begin()

// this function deletes a node from a particular location in the linked list
void list::delete_location(void)
{
    struct node *previous, *current;
    int pos = 0;
    // if the list is empty
    if(start==NULL)
    {
        cout<<"List is empty";
        getch();
        return;
    }
    else
    {
        int n;
        cout<<"Enter a value to search : ";
        cin>>n;

        // search the value in the list
        current = previous = start;
        while(current!= NULL)
        {
            if(current->data ==n)

```



```

        pos = 1;
        break;
    }
    previous = current;
    current = current->link;
}

// delete the node if value found in the list
if(pos == 1)
{
    previous->link = current->link;
    delete current;
    cout<<"Node is deleted";
}
else
    cout<<"Value not found :" <<endl;
getch();
}

} //end of delete_location()

// this function sorts a singly linked list using bubble method
void list::bubble_sort()
{
    if(start==NULL)
    {
        cout<<"List is empty";
        getch();
        return;
    }
    struct node *current, *previous;
    int size, temp;
    size = count(); // find the size of list
    // sort the linked list
    for(int u = size; u>1; u--)
    {
        current = start;
        for(int i = 1; i < u; i++)
        {
            previous = current;
            current = current->link;
            if(previous->data > current->data)
            {
                temp = previous->data;
                previous->data = current->data;
                current->data = temp;
            }
        } //end of inner for loop
    } //end of upper for loop
    getch();
} //end of bubble_sort()

```

DOUBLY LINKED LIST (TWO-WAY LIST)

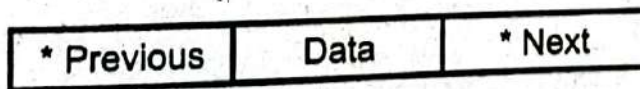
The linked list in which each node contains the addresses of both the next node and the previous node is called *doubly or double linked list*. This type of linked list can be visited or traversed in both directions i.e. in forward as well as in backward directions. Therefore, it is also called a *bi-directional linked list* or *two-way linked list*.

Node Structure of Doubly Linked List

The basic node structure of the doubly linked list consists of at least the following three fields:

- ★ First field contains the address of the previous node in the list. It is often called the *previous or backward link field*.
- ★ Second field contains the actual data or information of the node.
- ★ Third field contains the address of the next node in the list. It is often called the *next or forward link field*.

Following figure shows the basic structure of a node of the doubly linked list:



The first node and the last node of the doubly linked list contain the terminator, generally NULL value that determines the start and end of the list. Following figure shows the basic structure of a doubly-linked list consisting of three nodes:

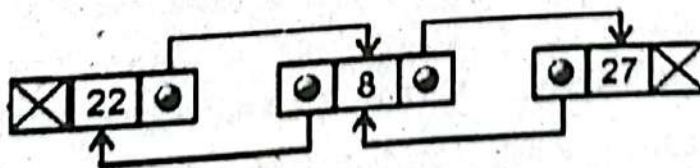


Figure: Doubly Linked List

3.1 Applications of Doubly Linked List

There are various applications of doubly linked list in the real world systems and computer applications. Some of them are as follows:

- Doubly linked list can be used in navigation systems where both forward and backward navigation is required.

- It is used by web browsers to implement backward and forward navigation of visited web pages by clicking BACK and FORWARD buttons. The browser cache stores the previous page and the next page.
- It is also used by various applications like MS Word to implement Undo and Redo functionality.
- A stack, hash table, and binary tree can be implemented using a doubly-linked list.
- It can be used to represent the *deck of cards* in games.
- It is also used to represent various states of a game.

9.3.2 Advantages and Disadvantages of Doubly Linked List

Following are some important advantages of a doubly-linked list:

- ★ We can traverse the doubly linked list in both directions i.e. from start to end and as well as from end to start. We know that backward traversal of nodes in a singly linked list is not possible.
- ★ It is the easiest data structure to implement.
- ★ Insertion and deletion operations are easier than those of a singly linked list. For example, to delete a node in a singly linked list, we need the pointer to its previous node. But in the case of a doubly-linked list, we just need the pointer to the node being deleted.
- ★ Reversing the list is simple and straightforward.
- ★ The doubly linked list can be used to represent other data structures as well. For example, a hierarchical structure can easily be represented using a doubly-linked list. Similarly, graphs can be represented using a doubly-linked list.

Following are some disadvantages of the doubly linked list:

- ❖ Each node of the doubly linked list requires extra space for a pointer to the previous node.
- ❖ Insertion and deletion operations take more time than a singly linked list.
- ❖ While manipulating the list, extra care is needed to modify both links of a node. For example, in the insertion of a node, we need to modify the previous pointer together with the next pointer.

9.3.3 Representation of a Doubly Linked List

A doubly linked list is represented in memory as a data structure having at least three fields. In C++, it is defined as follows:

```
struct node
```

```
{
```

```

node *previous;
int data;
node *next;
}*start, *prev, *temp;

```

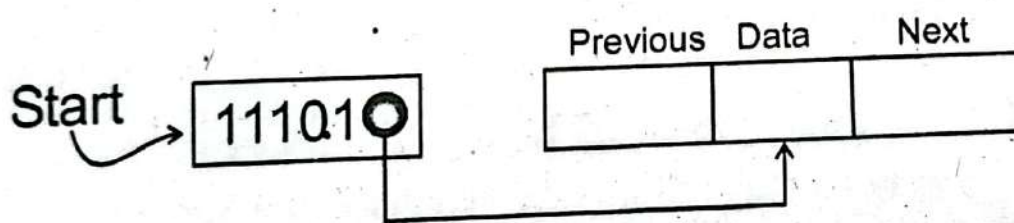
The variables 'start', 'prev' and 'temp' are pointer variables. The purpose of these pointer variables is as follows:

- ★ 'start' is used to hold the starting address of the list.
- ★ 'prev' is used to hold the address of the previous node of the list when a new node is created.
- ★ 'temp' is used to temporarily hold the address of the newly created node.

Suppose the first node of the list is created by using the new operator:

```
temp = new node;
```

The node is created in memory and its address is stored in 'temp'. Assume its memory address is 11101.



Now assign the value of 'temp' to 'start':

```
start = temp;
```

The contents of 'start' remain unchanged during program execution. To store the values in the above node, statements are written as follows:

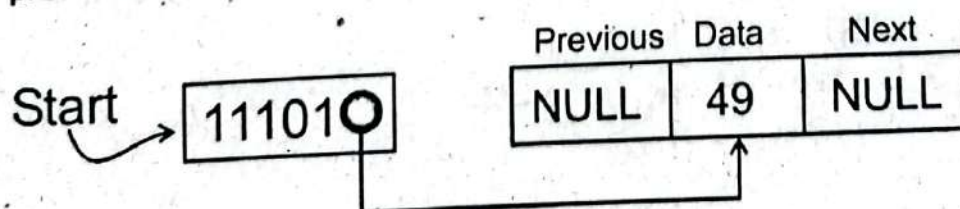
```
start->previous = NULL;
```

```
start->data = 49;
```

```
start->next = NULL;
```

```
prev = temp;
```

//Store 11101 in 'prev' i.e. previous address

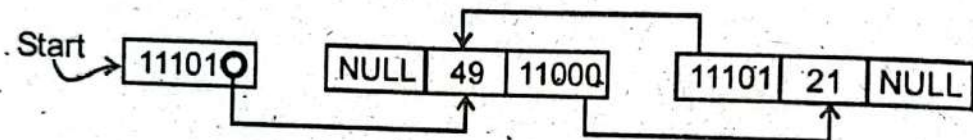


The values in the previous and next fields of the first node are NULL. It indicates that the list has only one node. The previous pointer field of the first node is always NULL for a double-linked list.

Following statements create a new node and connect (link) it to the list as a second node

```
// Create new node
temp = new node;
// Assign address of the newly created node to its preceding node
prev->next = temp;
// Store previous node address
temp->previous = prev;
// Assign values 21 to data field and NULL to next
temp->data = 21;
temp->next = NULL
```

Suppose the second node having value 21 is created at address 11000. The list with two nodes will be created as shown in the following figure:



Algorithm – Creating a Doubly Linked List

Write an algorithm that creates a doubly linked list which consists of 5 nodes. Suppose 'Data' is the data field of the node and Previous & Next are its pointer fields for storing addresses of previous and next nodes respectively.

1. START
2. Declare three pointer variables START, PREV, and TEMP of type node. Assign NULL to the START variable.
3. REPEAT Step-4 to Step-7 FOR C = 1 to 5
4. INPUT value for node in VAL
5. TEMP = new node
6. [Assign values to node TEMP]
 - TEMP->Data = VAL
 - TEMP->Next = NULL
7. IF START = NULL THEN
 - TEMP->Previous = NULL
 - START = TEMP
- ELSE
 - [go to the end of list]
 - CURRENT = START
 - WHILE CURRENT->Next != NULL

```
        CURRENT = CURRENT->Next
    END WHILE
    CURRENT->Next = TEMP
    TEMP->Previous = CURRENT
END IF
[End of step-3 loop]
8. EXIT
```

Program Code 9.3

//Program that creates a doubly linked list

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
struct node
{
    node *previous;
    int data;
    node *next;
};

class list
{
    private:
        node *start, *current, *temp;
    public:
        list()
        { start = NULL; }

        void add_item(int);
        void display(void);
};

void main (void)
{
    list obj;
    int val;
    clrscr();
    cout<<"Enter five values \n";
    for(int i = 1; i<=5; i++)
    {
        cin>>val;
        obj.add_item(val);
    }
    obj.display();
}
```



```

        getch();
    } //end of main()

    // Member function to add new data item in the list
    void list::add_item(int x)
    {
        // Create a node & assign data to it
        temp = new node;
        temp->data = x;
        temp->next = NULL;
        if(start == NULL)
        {
            temp->previous = NULL;
            start = temp;
        }
        else
            // go to end of list
        {
            current = start;
            while(current->next != NULL)
                current = current->next;
            current->next = temp;
            temp->previous = current;
        }
    } //end of add_item()

    // Member function to display data of double-linked list
    void list::display(void)
    {
        cout<<"Data in doubly linked list\n";
        current = start;
        while(current != NULL)
        {
            cout<<current->data<<endl;
            current = current->next;
        }
    } //end of display()

```

Output of the program:

Enter five values

25

14

98

63

75

Data in doubly linked list

25

14

98

63

75

9.3.4 Doubly Linked List Operations

Like a singly linked list, different operations can be performed on a doubly linked list. Some examples of these operations are traversing, insertion, deletion, searching, and sorting, etc. A brief description of important doubly linked list operations is as follows:

9.3.4.1 Traversing Doubly Linked List (DLL)

In a doubly-linked list, each node has two references, *next* and *previous* which make it possible to traverse the list in both directions (i.e. in forward as well as in backward directions). In forward traversal of the doubly linked list, we walk the list from the beginning and process each element until we reach the last node. Similarly, in the backward traversal of the doubly linked list, we walk the list from the last node and process each element until we reach the first node.

Algorithm – Forward Traversal of DLL

Write an algorithm that traverses the doubly linked list in the forward direction.

1. START
2. [Assign the starting address of the list to TEMP pointer variable]
TEMP = START
3. REPEAT step-4 to step-5 WHILE TEMP->next!=NULL
4. PRINT TEMP->data
5. TEMP = TEMP->next
[End of step-3 loop]
6. EXIT

forward_display() function

// this function displays the data of DLL in forward order
void forward_display(void)

```
{  
    struct node *current;  
    // if the list is empty  
    if(start==NULL)
```



```

    {
        cout<<"List is empty";
        return;
    }
    else
    {
        current = start;
        // display contents of list in forward order
        cout<<"Data of DLL In forward order \n";
        while(current!= NULL)
        {
            cout<<current->data<<endl;
            current = current->next;
        }
    }
} //end of forward_display()

```

Algorithm – Backward Traversal of DLL

Write an algorithm that traverses the doubly linked list in a backward direction.

1. START
2. Go to the end of the list and store the address of the last node of the list into the TEMP pointer variable
3. REPEAT step-4 to step-5 WHILE TEMP->previous!=NULL
4. PRINT TEMP->data
5. TEMP = TEMP->previous
[End of step-3 loop]
6. EXIT

backward_display() function

// this function displays the data of DLL in backward order

```

void backward_display(void)
{
    struct node *current;

    // if the list is empty
    if(start==NULL)
    {
        cout<<"List is empty";
        return;
    }
    else
    {

```

```

        // go to end of the list
        current = start;
        while(current->next!= NULL)
            current = current->next;

        // display contents of list in backward order
        cout<<"Data of DLL in backward order \n";
        while(current!= NULL)
        {
            cout<<current->data<<endl;
            current = current->previous;
        }
    }
} //end of backward_display()

```

Complexity Analysis of Traversal Algorithm of DLL

Like a singly linked list, a DLL can only be visited via its head node. All nodes of the list have to be accessed in the worst-case (from the head node to the last node one by one). Therefore, time complexity and space complexity of the DLL traversal algorithm are as follows:

Time Complexity		Space Complexity
Average-Case	Worst-Case	Worst-Case
$\Theta(n)$	$O(n)$	$O(n)$

9.3.4.2 Insertion in Doubly Linked List (DLL)

Like a singly linked list, the insertion can take place in a doubly-linked list at the following positions:

- i) At the beginning of the list
- ii) At the end of the list
- iii) At a specific location of a list

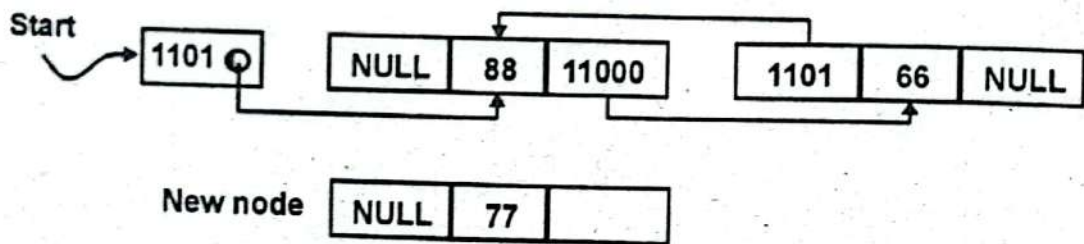
i) Insertion of a new node at the beginning in a DLL

When a new node is added before the head of the given doubly linked list, the newly added node becomes the new head of DLL. We create a new node and assign NULL to its previous pointer field, then we check whether the linked list is empty or not:

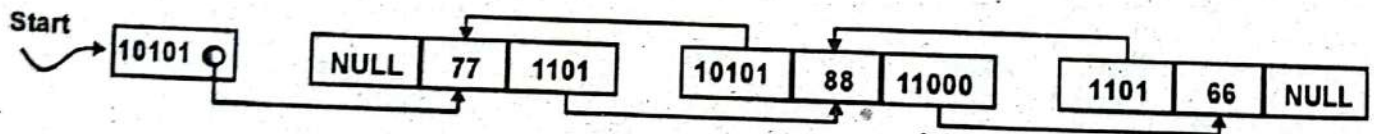
- If the linked list is empty, the new node is added as the first node of the list by performing the following steps:
 - assign a NULL value to the next pointer field of the new node.
 - assign the address of the new node to 'start' or 'head' pointer variable.

- If the linked list is not empty, then a new node is added before the head of the given linked list by performing the following steps:
 - assign the starting address of the list to the next pointer field of the new node.
 - assign the address of the new node to the previous pointer field of the node currently pointed by 'start' or 'head' variable i.e. first node of the existing list.
 - assign the address of the new node to the 'start' or 'head' pointer variable.

Suppose a linked list having two nodes and a newly created node are shown in the following figures:



Further, assume that the memory address of the newly created node is 10101 and a value 77 is assigned to its data field. The starting address of the list "1101" will be assigned to the next field of the newly created node. The address 10101 of the newly created node will be assigned to the previous field of the node pointed by the start. Then address 10101 of the newly created node will be assigned to the start variable as a starting address of the list.



Algorithm – Insertion at the beginning of a DLL

Write an algorithm that adds a new node at the beginning of a doubly linked list. Suppose the pointer variable "START" stores the starting address of the list.

1. START
2. Declare a pointer variable TEMP of type node.
3. Create a new node TEMP
4. [Assign a value X to the data field and NULL to the previous field of TEMP node.]
 TEMP->data = X
 TEMP->previous = NULL
5. IF the linked list is empty, THEN;
 [Assign a NULL value to the next field of the new node and address of the newly created node to the start variable.]
 TEMP->next = NULL
 START = TEMP
6. IF list is not empty, THEN;

[Assign the starting address of the list to the next pointer field of the new node and assign the address of the new node to the previous pointer field of the node currently pointed by the 'start' or 'head' variable. Finally, assign the address of the new node to the 'start' or 'head' pointer variable.]

```
TEMP->next = START
```

```
START->previous = TEMP
```

```
START = TEMP
```

7. EXIT

Insert_begin() function

// this function inserts a node at the beginning of a doubly linked list

```
void insert_begin(void)
```

```
{
```

```
    struct node *temp;
```

```
        // create a temp node in memory
```

```
    temp = new node;
```

```
        // input or assign data to temp node
```

```
    cout<<"Enter the data value for the node:";
```

```
    cin>>temp->data;
```

```
    temp->previous = NULL;
```

```
        // if the list is empty
```

```
    if(start==NULL)
```

```
{
```

```
        temp->next = NULL;
```

```
        start = temp;
```

```
}
```

```
else
```

```
        // if the list is not empty
```

```
{
```

```
    temp->next = start;
```

```
    start->previous = temp;
```

```
    start = temp;
```

```
}
```

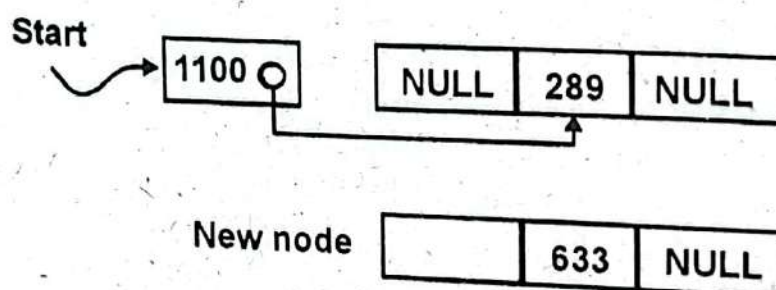
```
} //end of insert_begin()
```

Insertion of a new node at the end in a DLL

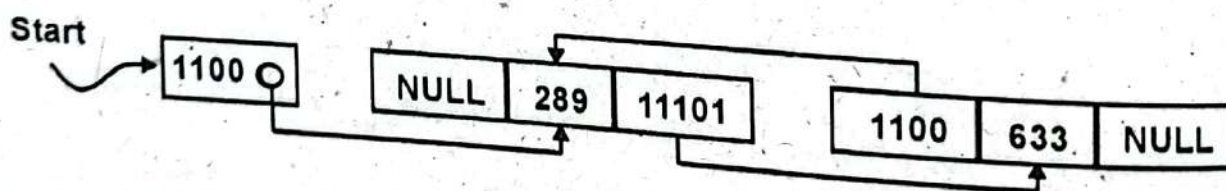
The insertion of a new node at the end of a doubly linked list is very simple. We create a new node and assign a NULL to its next pointer field, then we check whether the linked list is empty or not:

- If the linked list is empty, then we perform the following steps:
 - assign a NULL value to the previous pointer field of the new node.
 - assign the address of the new node to 'start' or 'head' pointer variable.
- If the linked list is not empty, then we perform the following steps:
 - go to the last node of the list.
 - assign the address of the new node to the next pointer field of the last node of the existing list.
 - assign the address of the last node of the existing list to the previous pointer field of the new node.

Suppose a linked list having one node and a newly created node are shown in the following figures:



Further, assume that the memory address of the newly created node is 11101 and a value 633 is assigned to its data field. The address of the new node i.e. 11101 will be assigned to the next field of the last node of the existing list and the address of the last node of the list i.e. 1100 will be assigned to the previous pointer field of the new node.



Algorithm – Insertion of a new node at the end of a doubly linked list

Write an algorithm that inserts a new node at the end of a doubly linked list. Suppose the pointer variable "START" stores the starting address of the list.

1. START
2. Declare two pointer variables TEMP and CURRENT of type node
3. Create a new node TEMP.
4. [Assign a value X to the data field and NULL to the next field of TEMP node.]
TEMP->data = X
TEMP->next = NULL
5. IF the linked list is empty, THEN;

[Assign a NULL value to the previous field of the new node and address of the newly created node to the start variable.]

TEMP->previous = NULL

START = TEMP

6. IF the list is not empty, THEN;

[Go to the last node of the list via CURRENT variable, assign the address of the new node to the next pointer field of the last node of the existing list, and then assign the address of the last node of the existing list to the previous pointer field of the new node.]

CURRENT->next = TEMP

TEMP->previous = CURRENT

7. EXIT

Insert_end() function

// this function inserts a new node at the end of a doubly linked list

void insert_end (void)

```
{
    struct node *temp, *current;

    // create a temp node in memory
    temp = new node;

    // input or assign data to temp node
    cout<<"Enter the data value for the node:";
    cin>>temp->data;
    temp->next = NULL;

    // if the list is empty
    if(start==NULL)
    {
        temp->previous = NULL;
        start = temp;
    }
    else
    {
        current = start;

        // go to end of the list
        while(current->next!= NULL)
            current = current->next;

        /* assign the address of the new node to the next field of the last node
        of the existing list and assign the address of the last node of the existing
        list to the previous pointer field of the new node. */
        current->next = temp;
        temp->previous = current;
    }
}
```



```

    }
    //end of insert_end()

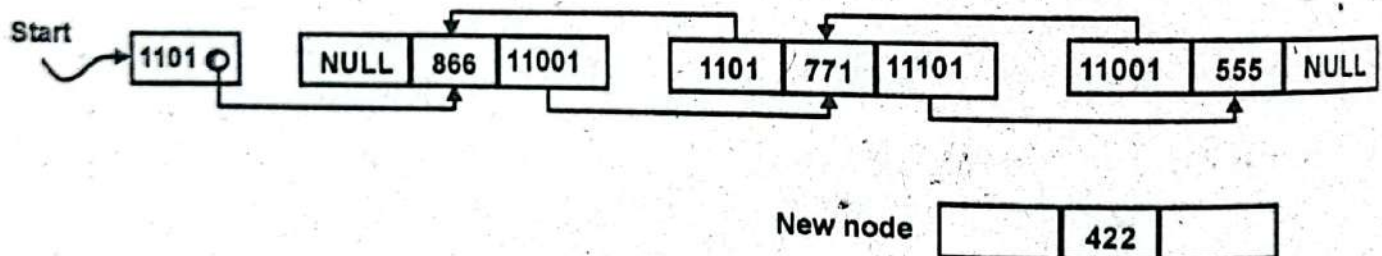
```

III) Insertion of a new node at a specific location in a DLL

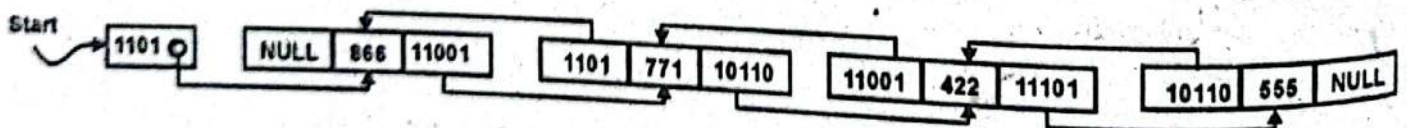
Like a singly linked list, we can insert a new node at a specific location in a doubly-linked list. Suppose we want to insert a new node after the second node of the list i.e. insertion between the second node and third node. We move to the second node of the list and set the values of the next and previous fields of the second node, new node, and third node as follows:

- **New Node:** The previous field will point to the second node and the next field will point to the third node of the list. The value of the previous node will be the address of the second node and the value of the next node will be the value of the next field of the second node.
- **Second Node:** The next field will point to the new node and the previous field will remain unchanged. The value of the next field of the second node will be the address of the new node.
- **Third Node:** The previous field will point to the new node and the next field will remain unchanged. The value of the previous field of the third node will be the address of the new node.

Suppose a linked list having three nodes and a newly created node are shown in the following figures:



Further, assume that the memory address of the newly created node is 10110 and a value 422 is assigned to its data field. We want to insert the new node after the second node that has value 771. The doubly linked list after inserting a new node is shown in the following figure:



Algorithm – Insertion at a specific location of Doubly Linked List

Write an algorithm that adds a new node at a specific location within a doubly linked list. Suppose the pointer variable "START" stores the starting address of the list.

1. START
2. Declare three pointer variables START, TEMP, and CURRENT of type node

3. Create a linked list with three nodes.
4. Create a new node TEMP
5. INPUT data value for TEMP in X
6. [Assign the value of X to the data field of the TEMP node.]
TEMP->data = X
7. IF the linked list is empty, THEN;
[Assign a NULL value to the previous field and next field of the new node. Then assign the address of the newly created node to the start variable.]
TEMP->next = NULL
TEMP->previous = NULL
START = TEMP
END IF
8. IF the list is not empty, THEN;
 - i) INPUT a value in Y to be searched in the list after which the new node is to be inserted.
 - ii) [Assign the value of starting address of the linked list into pointer variable CURRENT.]
CURRENT = START
 - iii) [Go to the node that has a value equal to Y via CURRENT pointer (i.e. after which the new node is to be inserted).]
WHILE CURRENT!=NULL
IF CURRENT->data = Y THEN
BREAK
END IF
CURRENT = CURRENT->next
END WHILE
 - iv) IF CURRENT = NULL, THEN
 - PRINT "Value not found"
 - EXIT
 - v) IF CURRENT->next == NULL i.e. searched value exists in the last node of the list THEN
TEMP->next = NULL
CURRENT->next = TEMP
TEMP->previous = CURRENT
 - vi) IF searched value found in other than the last node of the list THEN
 - [Assign the value of current node to the previous field of the new node and assign the value of next field of the current node to the next field of TEMP.]
TEMP->next = CURRENT->next
TEMP->previous = CURRENT

- [Assign the address of the new node to the previous field of the current node.]
CURRENT->next->previous = TEMP
- [Assign the address of the new node to the next field of the current node.]
CURRENT->next = TEMP

9. EXIT

Insert_location() function

// this function inserts a node at a particular location in the doubly linked list

```
void insert_location(void)
```

```
{
```

```
    struct node *temp, *current;
    int x, y;
```

```
        // create a temp node in memory
```

```
    temp = new node;
```

```
    cout<<"Enter the value to insert:";
```

```
    cin>>x;
```

```
    temp->data = x;
```

```
    if(start==NULL)
```

```
{
```

```
        temp->next = NULL;
```

```
        temp->previous = NULL;
```

```
        start = temp;
```

```
        return;
```

```
    }
```

```
    else
```

```
{
```

```
        cout<<"Enter the value to search:";
```

```
        cin>>y;
```

```
        // search the value 'y' in the list
```

```
        current = start;
```

```
        while(current!=NULL)
```

```
{
```

```
            if(current->data == y)
```

```
            break;
```

```
            current = current->next;
```

```
}
```

// if value found

// if value not found then insert the node after that node

```
if(current == NULL)
```

```

    {
        cout<<"Value not found in the list";
        return;
    }

    /* if value found and node is the last node of the list then insert the
    node after that node */
    if(current->next==NULL)
    {
        temp->next = NULL;
        current->next = temp;
        temp->previous = current;
    }
    else // if value found between nodes of the list
    {
        temp->next = current->next;
        temp->previous = current;
        (current->next)->previous = temp;
        current->next = temp;
    }
}
} //end of insert_location()

```

Complexity Analysis of Insertion Algorithm of DLL

Like the insertion algorithm of a singly linked list, the insertion algorithm of DLL also moves from the head node of the list, before the node where the new node is to be inserted. Then inserts a new node into the list. Thus the insertion algorithm of DLL takes $O(n)$ time for inserting a new node, where 'n' is the number of nodes in the list. Therefore, time complexity and space complexity of the insertion algorithm of DLL are as follows:

Time Complexity		Space Complexity
Average-Case	Worst-Case	Worst-Case
$O(n)$	$O(n)$	$O(n)$

3.4.3 Deletion In Doubly Linked List

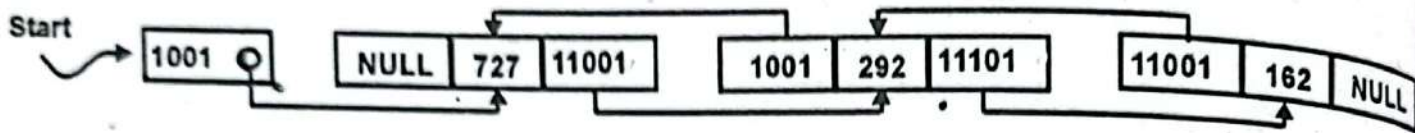
Like a singly linked list, deletion operation can be performed in a doubly-linked list at the following positions:

- At the beginning of the list
- At the end of the list
- After a particular location of a list

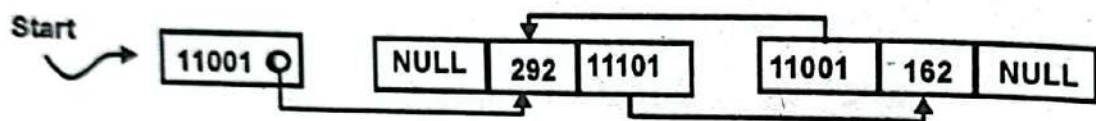
I) Deletion of a node from the beginning of DLL

To delete the first node of the DLL, we assign the value of the next pointer field of the first node of the list to the variable "start" (which holds the starting address of the list) and assign NULL to the previous pointer field of the second node.

Suppose a doubly linked list having three nodes is shown in the following figure:



After deleting the first node, the linked list will become as shown in the following figure:



Algorithm – Deleting a node from the beginning of a DLL

Write an algorithm that deletes a node from the beginning of a doubly linked list. Suppose the pointer variable "START" stores the starting address of the list.

1. START
2. Declare a pointer variable and TEMP of type node.
3. Create a doubly linked list.
4. [Assign the starting address of the list to the TEMP pointer variable.]
TEMP = START
5. [Assign the value of the next field of TEMP (address of the second node) to the START variable i.e. move to the second node.]
START = START->next
6. [Assign NULL to the previous pointer field of the second node.]
START->previous = NULL
7. [Delete the TEMP.]
Delete TEMP
8. EXIT

delete_begin() function

```

// this function deletes the first node of a doubly linked list
void delete_begin(void)
{
    struct node *temp;
  
```

```

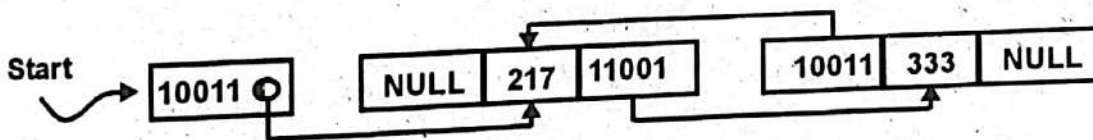
if(start==NULL)
{
    cout<<"List is empty";
    return;
}
else
{
    temp = start;
    cout<<"Data of first node : "<<temp->data;
    start = start->next;
    start->previous = NULL;
    delete temp;
    cout<<endl<<"First node is deleted";
}
} //end of delete_begin()

```

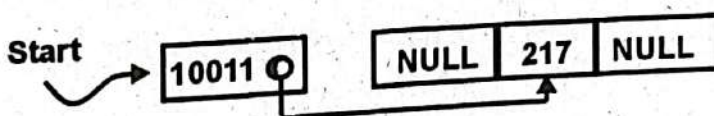
Deletion of a node from the end of a DLL

To delete a node from the end of the list, we move to the last node of the list and delete it. Then we move to the second last node of the list and assign NULL to its next pointer field.

Suppose a doubly linked list having two nodes is shown in the following figure:



After deleting the last node, the linked list will become as shown below:



Algorithm – Deleting a node from the end of a DLL

Write an algorithm that deletes a node from the end of a doubly linked list. Suppose the pointer variable "START" stores the starting address of the list.

1. START
2. Declare two pointer variables TEMP and CURRENT of type node.
3. Create a linked list.
4. [Go to the last node of the list.]
 CURRENT = START
 WHILE CURRENT->next != NULL

CURRENT = CURRENT->next

END WHILE

5. [Assign the address of the last node of the list to a temporary pointer variable.]
TEMP = CURRENT
6. [Move to the second last node of the list and assign NULL to its next pointer field.]
CURRENT = CURRENT->previous
CURRENT->next = NULL
7. [Delete the TEMP (i.e. previous last node).]
delete TEMP
8. EXIT

delete_end() function

// this function deletes a node from the end of a doubly linked list

```
void delete_end (void)
{
    struct node *temp, *current;

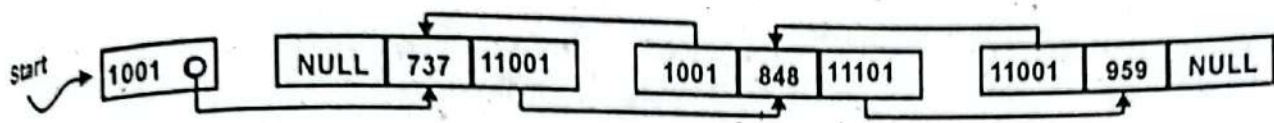
    // if the list is empty
    if(start==NULL)
    {
        cout<<"List is empty";
        return;
    }
    else
    {
        // go to end of the list
        current = start;
        while(current->next!= NULL)
            current = current->next;
        temp = current;
        cout<<"Data of last node is : "<<current->data;
        current = current->previous;
        current->next = NULL;
        delete temp;
        cout<<endl<<"Last node is deleted";
    }
} //end of delete_end()
```

iii) Deletion of a node after a particular Location in a DLL

Deleting a node after a particular location in a DLL is the same as deleting a node from the end, with a little difference. We move to the required node after which a node is to be deleted.

When we check whether the list is empty or the current node is the last node of the list. If not, then delete the node that exists after the current node.

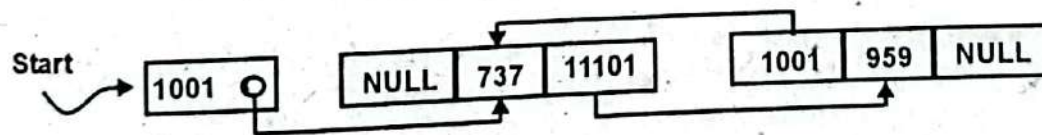
Suppose a linked list having three nodes is shown in the following figure.



Suppose we want to delete the node that exists after the first node of the list. In this case, we have to perform the following steps:

- move to the first node of the list that has value 737.
- assign the address of the second node (i.e. value of the next field of the first node which points to the second node) into a temporary pointer variable. The address of the second node is 11001.
- assign the value of the next field of the second node into the next field of the first node. The new value of the next field of the first node will be 11101.
- assign the address of the first field into the previous field of the third node (or assign the value of the previous field of the second node into the previous field of the third node). The address of the first node is 1001, so the new value of the previous field of the third node will be 1001.

After deleting the second node, the linked list will become as shown in the following figure:



Algorithm – Deletion of a node from a particular location within a DLL

Write an algorithm that deletes a node from a particular location in a doubly-linked list.

1. START
2. Declare two pointer variables TEMP and CURRENT of type node.
3. IF START = NULL THEN
 PRINT "List is empty"
 EXIT
END IF
4. INPUT a value in Y to be searched in the list after which a node is to be deleted.
5. [Go to the node that has value equal to Y via CURRENT pointer (i.e. node after which a node is to be deleted).]


```

CURRENT = START
WHILE CURRENT!=NULL
    IF CURRENT->data = Y THEN
        BREAK
    END IF
    CURRENT = CURRENT->next
END WHILE
6. IF CURRENT = NULL THEN
    PRINT "Value not found in the list"
    EXIT
END IF
7. IF CURRENT->next = NULL THEN
    PRINT "No node after current node"
    EXIT
ELSE
    TEMP = CURRENT->next
    CURRENT->next = TEMP->next
    (TEMP->next)->previous = CURRENT
    Delete TEMP
END IF
8. EXIT

```

delete_location () function

// this function deletes a node from a particular location in a doubly linked list

```

void delete_location (void)
{
    struct node *temp, *current;

    // if the list is empty
    if(start==NULL)
    {
        cout<<"The List is empty";
        return;
    }

    int n;
    cout<<"Enter a value to search :";
    cin>>n;

```

```

        // search the value in the list
        current = start;
        while(current != NULL)
        {
            if(current->data == n)
            {
                break;
            }
            current = current->next;
        }
        if(current == NULL)
        {
            cout<<"Value not found in the list:" <<endl;
            getch();
            return;
        }

        if(current->next == NULL)
        {
            cout<<"No node after the current node ";
            return;
        }
        else // delete the node if value found in the list
        {
            temp = current->next;
            current->next = temp->next;
            temp->next->previous = current;
            delete temp;
            cout<<"Node is deleted : " <<endl;
        }
    } //end of delete_location()

```

Complexity Analysis of Deletion Algorithm of DLL

Like the deletion algorithm of a singly linked list, the deletion algorithm of DLL moves from the head node of the list to the required node to be deleted. Then it deletes the required node from the list. Thus deletion algorithm of DLL takes $O(n)$ time for deleting a new node. Therefore, time complexity and space complexity of the deletion algorithm of DLL are as follows:

Time Complexity		Space Complexity
Average-Case	Worst-Case	Worst-Case
$\Theta(n)$	$O(n)$	$O(n)$

Program Code 9.4

// Program that implements the traversal, insertion and deletion operations of a DLL

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
struct node
```

```
{
```

```
    node *previous;
```

```
    int data;
```

```
    node *next;
```

```
};
```

```
class dll
```

```
{
```

```
    private:
```

```
        node *start;
```

```
    public:
```

```
        dll(void)
```

```
{
```

```
        start = NULL;
```

```
}
```

```
        // prototypes of member functions
```

```
        void insert_begin(void);
```

```
        void insert_end(void);
```

```
        void insert_location(void);
```

```
        void delete_begin(void);
```

```
        void delete_end(void);
```

```
        void delete_location(void);
```

```
        void forward_display(void);
```

```
        void backward_display(void);
```

```
};
```

```
void main(void)
```

```
{
```

```
    dll obj;
```

```
    int op, loop = 1;
```

```
    while(loop)
```

```
{
```

```
        clrscr();
```

```
        cout<<"1-Insert node at the beginning"<<endl;
```

```
        cout<<"2-Insert node at the end"<<endl;
```

```
        cout<<"3-Insert node at a specific location"<<endl;
```

```
        cout<<"4-Delete node from the beginning"<<endl;
```

```
        cout<<"5-Delete node from the end"<<endl;
```

```
        cout<<"6-Delete node from a specific location"<<endl;
```

```
cout<<"7-Display data of nodes in forward direction"<<endl;
cout<<"8-Display data of nodes in backward direction"<<endl;
cout<<"9- Exit"<<endl;
cout<<"Enter your option [1-9]";
cin>>op;
switch(op)
{
```

```
    case 1:
        obj.insert_begin();
        break;
    case 2:
        obj.insert_end();
        break;
    case 3:
        obj.insert_location();
        break;
    case 4:
        obj.delete_begin();
        break;
    case 5:
        obj.delete_end();
        break;
    case 6:
        obj.delete_location();
        break;
    case 7:
        obj.forward_display();
        break;
    case 8:
        obj.backward_display();
        break;
    case 9:
        loop = -1;
        break;
    default:
        cout<<"Invalid option"; break;
```

```
    } //end of switch
    if(loop == -1) break;
```

```
} //end of while loop
```

```
} //end of main ()
```

```
// this function inserts a node at the beginning of a doubly linked list
```

```
void dll::insert_begin(void)
{
```

```
    struct node *temp;
```



```

        // create a temp node in memory
temp = new node;

        // input or assign data to temp node
cout<<"Enter the data value for the node:";
cin>>temp->data;
temp->previous = NULL;

        // if the list is empty
if(start==NULL)
{
    temp->next = NULL;
    start = temp;
}
else
    // if the list is not empty
    {
        temp->next = start;
        start->previous = temp;
        start = temp;
    }
} //end of insert_begin()

// this function inserts a new node at the end of a doubly linked list
void dll::insert_end (void)
{
    struct node *temp, *current;

        // create a temp node in memory
temp = new node;

        // input or assign data to temp node
cout<<"Enter the data value for the node:";
cin>>temp->data;
temp->next = NULL;

        // if the list is empty
if(start==NULL)
{
    temp->previous = NULL;
    start = temp;
}
else
{
        // go to end of the list
current = start;
while(current->next!= NULL)
    current = current->next;

```

```
/* assign the address of the new node to the next field of the last node
of the existing list and assign the address of the last node of the existing
list to the previous pointer field of the new node. */
```

```
current->next = temp;
temp->previous = current;
```

```
}
```

```
} //end of insert_end()
```

```
// this function inserts a node at a particular location in the doubly linked list
```

```
void dll::Insert_location(void)
```

```
{
```

```
    struct node *temp, *current;
```

```
    int x, y;
```

```
        // create a temp node in memory
```

```
    temp = new node;
```

```
    cout<<"Enter the value to insert:";
```

```
    cin>>x;
```

```
    temp->data = x;
```

```
    if(start==NULL)
```

```
    {
```

```
        temp->next = NULL;
```

```
        temp->previous = NULL;
```

```
        start = temp;
```

```
        return;
```

```
    }
```

```
    else
```

```
    {
```

```
        cout<<"Enter the value to search:";
```

```
        cin>>y;
```

```
        // search the value 'y' in the list
```

```
        current = start;
```

```
        while(current!= NULL)
```

```
        {
```

```
            if(current->data == y)
```

```
            // if value found
```

```
            break;
```

```
            current = current->next;
```

```
        }
```

```
        // if value not found then insert the node after that node
```

```
        if(current == NULL)
```

```
        {
```

```
            cout<<"Value not found in the list";
```

```
            getch();
```

```
            return;
```

```
        }
```



```

        /* if value found and node is the last node of the list then insert the
        node after that node */
        if(current->next==NULL)
        {
            temp->next = NULL;
            current->next = temp;
            temp->previous = current;
        }
        else // if value found between nodes of the list
        {
            temp->next = current->next;
            temp->previous = current;
            (current->next)->previous = temp;
            current->next = temp;
        }
    }
} //end of insert_location()

// this function deletes the first node of a doubly linked list
void dll::delete_begin(void)
{
    struct node *temp;
    if(start==NULL)
    {
        cout<<"List is empty";
        getch();
        return;
    }
    else
    {
        temp = start;
        cout<<"Data of first node : "<<temp->data;
        start = start->next;
        start->previous = NULL;
        delete temp;
        cout<<endl<<"First node is deleted";
        getch();
    }
} //end of delete_begin()

// this function deletes a node from the end of a doubly linked list
void dll::delete_end (void)
{
    struct node *temp, *current;
    // if the list is empty
    if(start==NULL)

```

```
{
    cout<<"List is empty";
    getch();
    return;
}
else
{
    // go to end of the list
    current = start;
    while(current->next!= NULL)
        current = current->next;
    temp = current;
    cout<<"Data of last node is:"<<current->data;
    current = current->previous;
    current->next = NULL;
    delete temp;
    cout<<endl<<"Last node is deleted";
    getch();
}
} //end of delete_end()
```

// this function deletes a node from a particular location in a doubly linked list

```
void dll::delete_location (void)
```

```
{
    struct node *temp, *current;

    // if the list is empty
    if(start==NULL)
    {
        cout<<"List is empty";
        getch();
        return;
    }

    int n;
    cout<<"Enter a value to search :";
    cin>>n;

    // search the value in the list
    current = start;
    while(current!= NULL)
    {
        if(current->data ==n)
            break;
        current = current->next;
    }

    if(current== NULL)
```



```

    {
        cout<<"Value not found in the list:" <<endl;
        getch();
        return;
    }

    if(current->next== NULL)
    {
        cout<<"No node after current node ";
        getch();
        return;
    }
    else // delete the node if value found in the list
    {
        temp = current->next;
        current->next = temp->next;
        temp->next->previous = current;
        delete temp;
        cout<<"Node is deleted ." <<endl;
        getch();
    }
} // end of delete_location()

// this function displays the data of DDL in forward order
void dll::forward_display(void)
{
    struct node *current;

    // if the list is empty
    if(start==NULL)
    {
        cout<<"List is empty";
        getch();
        return;
    }
    else
    {
        current = start;

        // display contents of list in forward order
        cout<<"Data of DDL In forward order \n";
        while(current!= NULL)
        {
            cout<<current->data<<endl;
            current = current->next;
        }
    }
}

```

```

    getch();
} //end of forward_display()

// this function displays the data of DLL in backward order
void dll::backward_display(void)
{
    struct node *current;

    // if the list is empty
    if(start==NULL)
    {
        cout<<"List is empty";
        getch();
        return;
    }
    else
    {
        // go to end of the list
        current = start;
        while(current->next!= NULL)
            current = current->next;

        // display contents of list in backward order
        cout<<"Data of DDL in backward order \n";
        while(current!= NULL)
        {
            cout<<current->data<<endl;
            current = current->previous;
        }
    }
    getch();
} //end of backward_display()

```

9.3.5 Differences between Doubly Linked List and Singly Linked List

The main differences between a doubly linked list and singly linked list are given in the following table:

Singly Linked List	Doubly Linked List
1- Each node contains only one pointer field that points to the next node of the list.	1- Each node contains two pointer fields. the first points to the next node, and the second points to the previous node of the list.
2- Traversing can be done in only one direction.	2- Traversing can be done in both directions.

3- It uses less memory ^a per node (single pointer).	3- It uses more memory per node (two pointers).
4- It can mostly be used to implement stacks.	4- It can be used to implement stacks, heaps, binary trees.

9.4 CIRCULARLY LINKED LISTS

In a singly linked list, the last node has a *NULL* value in its link or pointer field. It indicates that the last node does not point to any next node or element of the list. The last node of a linked list can be used to point back to the first node. In this situation, all nodes are linked in a continuous circle and form a circular chain, without using *NULL*. This type of linked list is called a *circularly linked list* or *circular linked list*. Both *singly linked list* and *doubly linked list* can be made the circular linked lists.

In a circular linked list, the previous node stores the address of the next node, and the last node stores the address of the first node. The end of the list is detected with reference to the pointer field of the nodes. If the pointer field value of any node in the list points to the starting node (first node) of the list, that node will be the last node of the list. A circular singly linked list having four nodes is shown in the following figure:

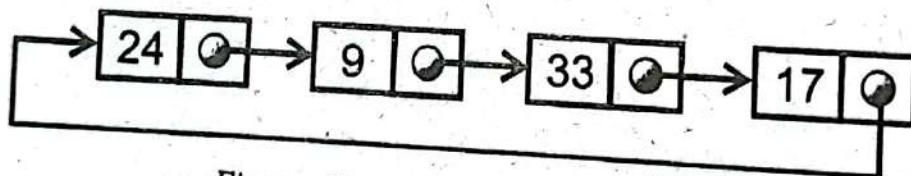


Figure: Circular Singly Linked List

Suppose the starting address of the list is 1011 that points to the first node. The node 4 has the same value i.e. 1011 in its pointer field. It points to the first node. Therefore, it is the last node of the circular linked list.

9.4.1 Application of Circular Linked List

Circular linked lists are used in certain applications that require the last node pointing back to the first node (i.e. applications that need repeatedly go around the list). Some important applications of circular linked lists are as follows:

- **Operating Systems:** In personal computers, operating systems use circular linked lists to handle jobs or tasks. For example, round-robin time sharing jobs of the operating system or simple multitasking in a PC system uses a circular linked list technique to handle jobs or tasks. All the running applications are kept in a circular linked list and the operating system gives a fixed time of slice to all the jobs for running. The operating system keeps on iterating over the linked list until all the applications are completed.
- **Multiplayer Games:** In multiplayer games, all the players are kept in a circular linked list. After a player has taken his turn, it advances to the next player in the list and this process continues in a circular fashion.

- **Implementation of Queue:** Circular linked list is useful for the implementation of a queue (simple queue and circular queue). In a queue, two pointers, FRONT, and REAR are required, whereas, in a circular linked list, only one pointer is required.

Operations on Circular Linked List

The operations on the circular linked list are similar to those of the linear linked list.

Algorithm – Circular Linked List

Write an algorithm to create a circular linked list consisting of ten nodes.

1. Define the structure of a node of the list
2. Define Pointer Variables S(for START), P(for PREVIOUS), and C (for CURRENT) of the list
3. Create the first node in P and assign its address to S
 $S = P$
4. REPEAT Step-5 to 7 FOR I = 1 to 10
5. Create a new node in C and assign value to it
6. Assign address of C to the previous node
 $P \rightarrow \text{link} = C$
7. Assign address of C to P
 $P = C$
 [End of Step-4 loop]
8. Assign the address of the starting node in the pointer field of the last node
 $C \rightarrow \text{link} = S$
9. EXIT

Program Code 9.5

// Program that creates a circular singly linked list having 5 integer values

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
struct node
```

```
{
```

```
    int n;
```

```
    node *link;
```

```
};
```

```
class list
```

```
{
```

```
    private:
```

```
        node *S,*C,*P;
```

```
    public:
```



```

        list(void)
        {
            S = NULL;
        }

        void insert(int);
        void display(void);
    };

main()
{
    list obj;
    int x;
    clrscr();
    cout<<"Enter 5 integer values\n";
    for(int i = 1; i<=5; i++)
    {
        cin>>x;
        obj.insert(x);
    }
    obj.display();
    getch();
} //end of main()

// member function to add new node in a list
void list::insert(int data)
{
    // to create first node and assign data to it
    if(S == NULL)
    {
        P = new node;
        P->n = data;
        S = P;
    }
    else
    {
        C = new node;
        C->n = data;
        P->link = C;
        P = C;
    }
} //end of insert()

// member function to display values
void list::display(void)
{
    C = S;

```

```
cout<<"Values of the list : "<<endl;
while(1)
```

```
{
    cout<<C->n<<endl;
    if(C==P) break;
    C = C->link;
```

```
} //end of display()
```

Output of the program:

Enter 5 integer values

25

14

85

96

72

Values of list:

25

14

85

96

72

4.2 Circularly Double-Linked Lists

In a circularly double-linked list, each node has two pointer fields, similar to a *doubly-linked list*, except that the previous pointer field of the first node points to the last node and the next pointer field of the last node points to the first node.

A circularly double-linked list with three nodes is shown graphically in the following figure:

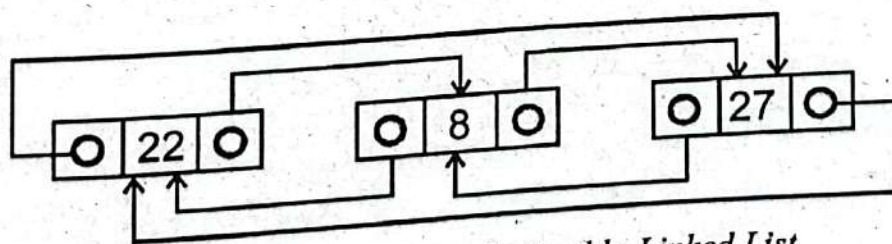
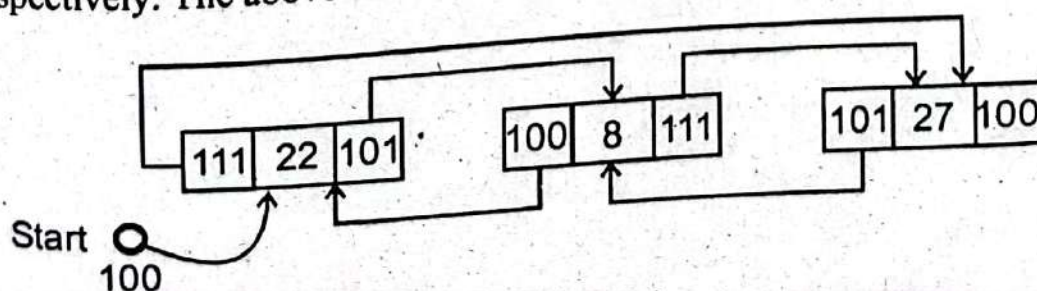


Figure: Circularly Double-Linked List

Suppose a circularly double-linked list with three nodes is created having addresses 100, 111, and 111 respectively. The above-linked list with addresses is shown in the following figure:



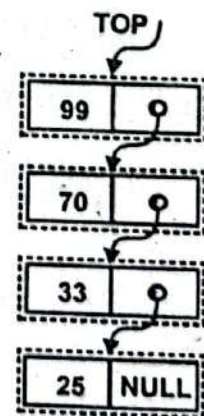
9.5 STACK IMPLEMENTATION USING LINKED LIST

The stack implementation using array has already been discussed in chapter 4. The major problem or disadvantage of stack implementation using an array is that its size remains fixed because it needs to be specified at compile time. Therefore such stack implementation is not very useful (i.e. not suitable) when we are unaware of the size of the data which we are going to use at runtime.

However, a stack data structure can also be implemented using a linked list data structure. We have already learned the concept of linked list in part-1 of this chapter. A linked list does not have any restrictions on the number of nodes or elements it can hold. Memory space for the elements in a linked list is allocated dynamically. Hence the linked list can grow as long as there is enough memory available for dynamic allocation. Therefore, stack implementation using a linked list can work for a variable number of elements. It means that the size of the stack can shrink or grow on demand during the program execution. Stack implementation using a linked list is also known as a *linked stack*.

In linked stack, every new node or element is inserted as a 'top' element. That means every newly inserted node is pointed by 'top'. Whenever we want to remove a node from the stack, simply remove the node which is pointed by 'top' by moving 'top' to its next node in the list. The next field of the first node must always be a NULL value.

Before the implementation of actual stack operations using a linked list, we define the structure of node with at least two members or fields namely **data** and **next** or **link**. We also define a pointer variable "TOP" of type node. In the figure, four nodes are shown. The last inserted node has a value 99, while the first inserted node has a value 25. The order of nodes inserted is 25, 33, 70, and 99. The pointer "TOP" points the last inserted node i.e. node with value 99.



9.5.1 Operations on Stack using Linked List

The fundamental operations on a stack are *push* and *pop*. The push operation is used to insert a new node on the stack, while the pop operation is used to remove a node from the top of the stack. These operations have already been discussed in chapter 4.

Example 1: (Push operation)

Suppose we create a new node and its memory address is 11011. We assign value 75 to its data field and insert this new node into an empty stack. Thus, we assign the NULL to the next

field of the new node and assign the address of the new node to the TOP pointer. Following figure shows the stack with a single node:

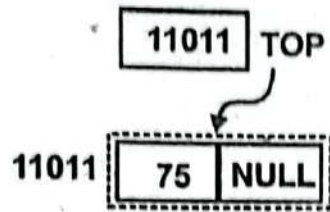
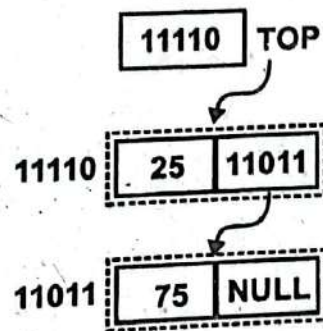


Figure (a)

Now we create another new node, suppose its memory address is 11110. We assign value 25 to its data field and insert this new node into the above non-empty stack by performing the following steps:

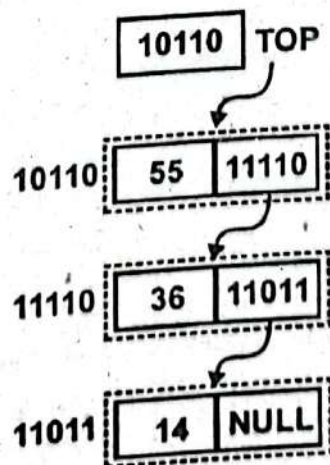
- assign the value of TOP to the next field of the new node.
- assign the address of the new node to the pointer TOP.

Following figure shows the stack (with two nodes) after adding a new node of address 11110:

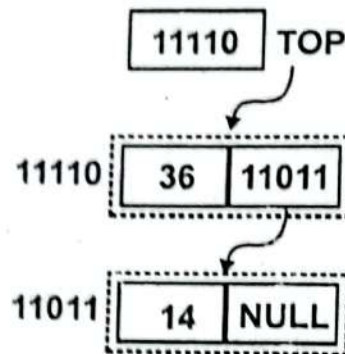


Example 2: (Pop operation)

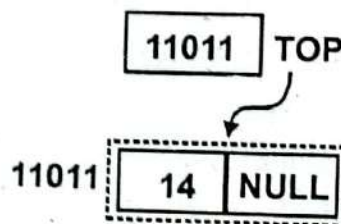
Following figure shows a stack with three nodes:



Before deleting the top node with value 55, we assign the value of the next field of this node to pointer variable TOP (i.e. $TOP = TOP \rightarrow next$), and then we delete this node. The value of the TOP variable becomes 11110. Following figure shows the stack after deleting the top node:



Following figure shows the stack after deleting the top node (with value 36) from the above figure:



Algorithm – Push Operation

Write an algorithm that inserts a new node on the stack using a linked list.

- 1- START
- 2- Create a new node with a given value.
- 3- Check whether stack is empty ($TOP == NULL$)
- 4- If the stack is empty, then assign a NULL value to the next field of the new node.
 $new_node \rightarrow next = NULL$
- 5- If the stack is not empty, then assign the address of TOP to the next field of the new node.
 $new_node \rightarrow next = TOP$
- 6- Finally assign the address of the new node to variable TOP.
 $TOP = new_node$
- 7- EXIT

Algorithm – Pop Operation

Write an algorithm that removes a node from a stack using linked list.

- 1- START
- 2- Check whether stack is empty ($TOP == NULL$).
- 3- If the stack is empty, then display "Stack is empty" and exit.
- 4- If the stack is not empty, then define a pointer 'TEMP' of type node and assign the value of 'TOP' to it.
 $TEMP = TOP$

- 5- Assign the value of the next field of node TOP to TOP.
TOP = TOP->next
- 6- Delete the 'TEMP'
- 7- EXIT

Program Code 9.6

// Program that implements the stack using linked list

```
#include <iostream.h>
#include <conio.h>

struct node
{
    int data;
    struct node *next;
};

class stack
{
private:
    struct node *top;
public:
    stack(void)
    {
        top = NULL;
    }
    void push(void);
    void pop(void);
    void display(void);
};

void main()
{
    stack obj;
    int opt, ok = 1;
    do
    {
        clrscr();
        cout<<"1. Push data into stack"<<endl;
        cout<<"2. Pop data from stack"<<endl;
        cout<<"3. Display data of stack"<<endl;
        cout<<"4. exit"<<endl;
        cout<<"Enter your option:[1-4] ? ";
        cin>>opt;
        switch(opt)
        {
            case 1 : obj.push(); break;
```



```

        case 2 : obj.pop(); break;
        case 3 : obj.display(); break;
        case 4 : ok = 0;
    } //end of switch
} while(ok);
} //end of main()

```

```

// member function to push value into stack
void stack::push(void)
{

```

```

    struct node *temp;
    temp = new node;
    cout<<"Enter data for node: ";
    cin>>temp->data;
    if(top == NULL)
        temp->next = NULL;
    else
        temp->next = top;
    top = temp;
    cout<<"Data is inserted "<<endl;
    getch();
} //end of push()

```

```

// member function to pop value from stack
void stack::pop(void)
{

```

```

    if( top == NULL)
    {
        cout<<"Stack is empty";
        getch();
        return;
    }
    cout<<"Data of node is : "<<top->data<<endl;
    struct node *temp;
    temp = top;
    top = top->next;
    delete temp;
    cout<<"This data is deleted "<<endl;
    getch();
} //end of pop()

```

```

// member function to display values of stack
void stack::display(void)
{

```

```

    if(top==NULL)
    {

```

```

        cout<<"Stack is empty"<<endl;
        getch();
        return;
    }
    else
    {
        struct node *temp;
        temp = top;
        cout<<"Data of stack is : "<<endl;
        while(temp!=NULL)
        {
            cout<<temp->data<<endl;
            temp = temp->next;
        }
    }
    getch();
} //end of display()

```

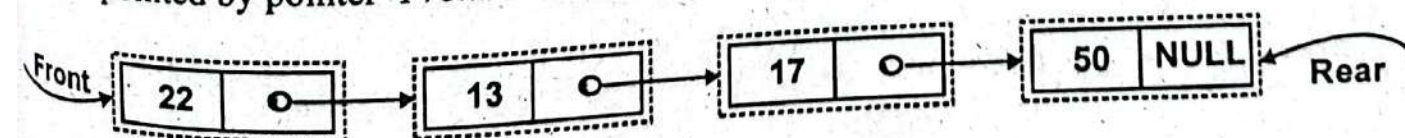
9.6 QUEUE IMPLEMENTATION USING LINKED LIST

A queue is a data structure, that uses the First in First Out (FIFO) principle. It means that the data item that is added at first into the queue is removed first. Similarly, the data item that is added at the last is removed at last.

Like stack, we can implement a queue using a linked list. The advantage of using a linked list is that there is no size limit. The size of the queue grows or shrinks as items (new nodes) are inserted (added) or nodes are removed from the queue.

In a queue data structure, we maintain two pointers, *Front* and *Rear*. The *Front* always points to the first element of the queue and the *Rear* points to the last element. Initially, we assign NULL to both *Front* and *Rear* pointers which indicate that the queue is empty. If the queue has only one element, then both *Front* and *Rear* will point to that single element.

Before implementation of a queue using a linked list, we define the structure of node with at least two members or fields namely **data** and **next**. We also define pointer variables "*Front*" and "*Rear*" of type node. Following figure shows a queue with four elements or nodes. The last inserted node has value 50 and it is pointed by pointer '*Rear*'. The first inserted node has value 22 and it is pointed by pointer '*Front*'. The order of nodes inserted is 22, 13, 17, and 50.



9.6.1 Operations on Queue using Linked List

The two basic operations *Insertion* (enqueue) and *Deletion* (dequeue) are performed on the queue. The insertion operation is performed at the *Rear* or *Back* and the deletion operation is performed at the *Front* end.

Example 1: (Insertion of new nodes)

Suppose we create a new node and its memory address is 10001. We assign value 88 to its data field and NULL to its next field. We add this new node to an empty queue. Thus, both Rear and Front pointers are assigned the address of the newly created node. Following figure shows the queue with this single node:

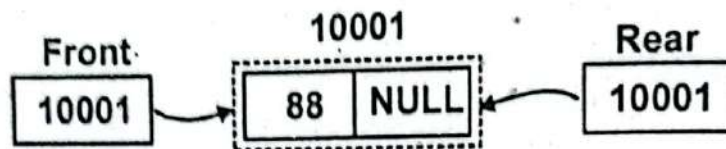
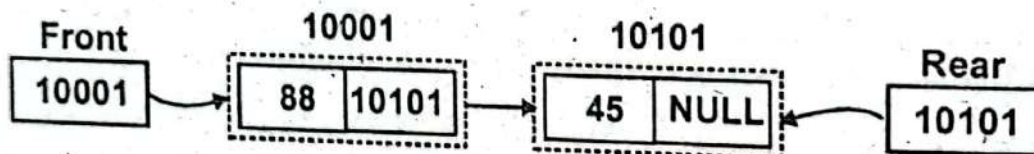


Figure (a)

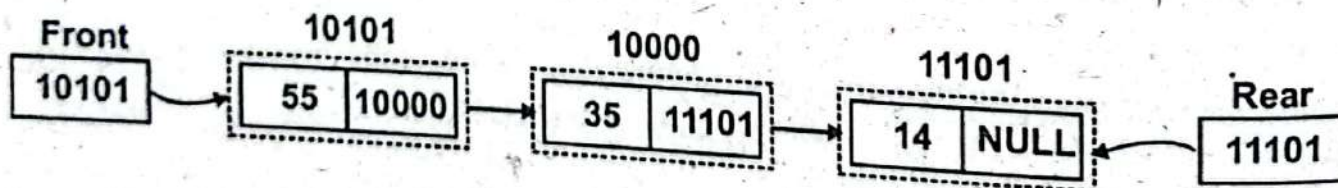
Now we create another new node, suppose its memory address is 10101. We assign value 45 to its data field and NULL to its next field. We add this new node to the non-empty queue by performing the following steps:

- assign the address of the new node to the next field of Rear.
- assign the address of the new node to Rear.

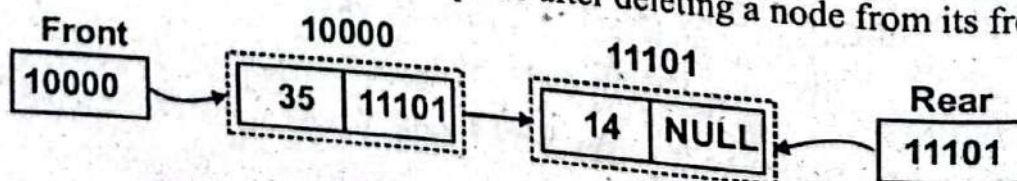
Suppose we add this node shown in the above figure (a). Following figure shows the queue (with two nodes) after adding a new node of address 10101:

**Example 2: (Deletion of nodes)**

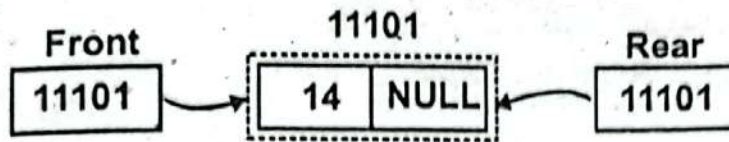
Following figure shows a queue with three nodes:



Before deleting the node with value 55, we assign the value of the next field of this node to the pointer variable Front, and then we delete this node. The value of the Front variable becomes 10000. Following figure shows the queue after deleting a node from its front end:



Following figure shows the queue after deleting a node with value 35:



Note that when a queue has only one node, the value of the Front pointer will be equal to the Rear pointer. After deleting the last node of the queue, both the Front and Rear pointers are assigned NULL.

Algorithm – Enqueue

Write an algorithm that inserts a new node into a queue using a linked list.

- 1- START
- 2- Create a new node, assign a value to its data field, and assign NULL to its next field.


```

      new_node->data = value
      new_node->next = NULL
      
```
- 3- Check whether the queue is empty (Rear == NULL)
- 4- If the queue is empty, then assign the address of the new node to both Rear and Front.


```

      Rear = Front = new_node
      
```
- 5- If the queue is not empty, then assign the address of the new node to the next field of Rear and then the address of the new node to Rear.


```

      Rear->next = new_node
      Rear = new_node
      
```
- 6- EXIT

Algorithm – Dequeue

Write an algorithm that removes a node from a queue using a linked list.

- 1- START
- 2- Check whether the queue is empty (Front == NULL).
- 3- If the queue is empty, then display "Queue is empty" and exit.
- 4- If the queue is not empty, then perform the following steps:
 - Define a pointer 'TEMP' of type node and assign the address of 'Front' to 'TEMP'.


```

          TEMP = Front
          
```
 - IF Front = Rear THEN // If queue has only one value


```

          Front = Rear = NULL
          
```
 - ELSE


```

          Front = Front->next
          
```
 - END IF
 - Delete the 'TEMP'
- 5- EXIT

Algorithm – Displaying the elements of the queue

Write an algorithm that displays the values of nodes of a queue using a linked list.

- 1- START
- 2- Check whether the queue is empty (Front == NULL).
- 3- If the queue is empty, then display "Queue is empty" and exit.
- 4- If the queue is not empty, then define a pointer 'TEMP' of type node and assign the address of 'Front' to 'TEMP'.
 TEMP = Front
- 5- Display the data of nodes of queue via TEMP variable until 'TEMP' reaches to 'Rear' (i.e. TEMP->next != NULL).
 WHILE TEMP->next != NULL
 PRINT TEMP->data
 TEMP = TEMP->next
 END WHILE
- 6- EXIT

Program Code 9.7

// Program that implements the queue using a linked list

```
#include <iostream.h>
#include <conio.h>
```

```
struct node
{
    int data;
    struct node *next;
};

class queue
{
private:
    struct node *front, *rear;
public:
    queue(void)
    {
        front = NULL;
        rear = NULL;
    }
    void enqueue(void);
    void dequeue(void);
    void display(void);
};
```

```
void main()
```

```
{
```

```
queue obj;  
int opt, ok = 1;  
do
```

```
{  
    clrscr();  
    cout<<"1. Insert data into queue "<<endl;  
    cout<<"2. Remove data from queue "<<endl;  
    cout<<"3. Display data of queue "<<endl;  
    cout<<"4. exit"<<endl;  
    cout<<"Enter your option:[1-4] ? ";  
    cin>>opt;  
    switch(opt)  
    {  
        case 1 : obj.enqueue(); break;  
        case 2 : obj.dequeue(); break;  
        case 3 : obj.display(); break;  
        case 4 : ok = 0;  
    } //end of switch
```

```
}while(ok);
```

```
} //end of main()
```

```
// member function to insert a value into queue
```

```
void queue::enqueue(void)
```

```
{  
    struct node *temp;  
    temp = new node;  
    cout<<"Enter data for node: ";  
    cin>>temp->data;  
    temp->next = NULL;  
    if(rear == NULL)  
    {  
        rear = front = NULL;  
        cout<<"Node is inserted as first node"<<endl;  
    }  
    else  
    {  
        rear->next = temp;  
        rear = temp;  
        cout<<"Node is inserted into queue"<<endl;  
    }  
    getch();  
} //end of enqueue()
```

```
// member function to remove a value from queue
```

```
void queue::dequeue(void)
```

```
{  
    if( front == NULL)
```



```

        cout<<"Queue is empty";
        getch();
        return;
    }
    cout<<"Data of node is : "<<front->data<<endl;
    struct node *temp;
    temp = front;
    if(rear == front)           // if queue has only one value
        rear = front = temp;
    else
        front = front->next;
    delete temp;
    cout<<"This node of queue is deleted "<<endl;
    getch();
} //end of dequeue()

// member function to display values of queue
void queue::display(void)
{
    struct node *temp;
    if(front==NULL)
    {
        cout<<"Queue is empty"<<endl;
        getch();
        return;
    }
    else
    {
        temp = front;
        cout<<"Data of queue is : "<<endl;
        while(temp!=NULL)
        {
            cout<<temp->data<<endl;
            temp = temp->next;
        }
    }
    getch();
} //end of display()

```

Source Code of Programs

Source code of programs given in "DATA STRUCTURES USING C++" can be received by sending an e-mail to "pakman_series@hotmail.com".

Summary --- Complexities of Linked List

Singly Linked List

Operation	Time Complexity		Space Complexity
	Average-Case	Worst-Case	Worst-Case
Traversal	$\Theta(n)$	$O(n)$	$O(n)$
Search	$\Theta(n)$	$O(n)$	$O(n)$
Insertion	$\Theta(n)$	$O(n)$	$O(n)$
Deletion	$\Theta(n)$	$O(n)$	$O(n)$

Doubly Linked List

Operation	Time Complexity		Space Complexity
	Average-Case	Worst-Case	Worst-Case
Traversal	$\Theta(n)$	$O(n)$	$O(n)$
Search	$\Theta(n)$	$O(n)$	$O(n)$
Insertion	$\Theta(n)$	$O(n)$	$O(n)$
Deletion	$\Theta(n)$	$O(n)$	$O(n)$

Short Answers to the Questions

What is a linked list?

Linked list is a linear data structure. It is a list or chain of *nodes* where each node stores data as well as points to the next node in the list. It means that each node contains the data and also the location of the next node. Thus each node of the list is linked logically with the next node through a link-pointer. The *head* of a linked list is its first node. The *tail* of a linked list refers to the last node in the list. The last node has a Null reference.

Define the singly linked list.

A type of linked list in which each node contains the data of node and the address of the next node only is called a *singly linked list*. Sometimes, it is also simply called a *single linked list*.

Define the circularly linked list.

A type of linked list in which all nodes are linked in a continuous circle, without using *null*, is called a *circularly linked list* or *circular linked list*.

What do you know about doubly linked list?

A type of linked list in which each node contains the addresses of both the next node and of the previous node is called a *doubly or double linked list*. This type of linked list can be visited in both directions, i.e. in forward and backward directions. Therefore it is also called a *two-way list*.

-  What is the difference between a singly linked list and a doubly-linked list?
- A singly linked list can be visited in one direction only, i.e. starting from the beginning towards the end of the list. The previous node cannot be accessed from the current node. A double-linked list, however, can be visited in both directions. This is the major difference between a simple linked list and a double-linked list.
-  What is circularly double linked list?
- In a circularly double-linked list, each node has two links, similar to a doubly-linked list, except that the previous link of the first node points to the last node and the next link of the last node points to the first node.

EXERCISE

Q#1 Select the correct answer from the multiple choices.

1. Which of the following is true about a linked list?
 - (a) It is a linear data structure
 - (b) It is a chain of nodes
 - (c) It is a list of nodes
 - (d) All of these
2. The head of the linked list is its:
 - (a) first node
 - (b) last node
 - (c) middle node
 - (d) None of these
3. In a linked list, the next part of a node is known as:
 - (a) next link
 - (b) next pointer
 - (c) linker
 - (d) Both a & b
4. Singly linked list is also called:
 - (a) single linked list
 - (b) one-way list
 - (c) one-way chain
 - (d) All
5. In a singly linked list, the value of the next link field of the last node is always:
 - (a) 0
 - (b) NULL
 - (c) NIL
 - (d) All
6. Which of the following operations can be performed on the linked list?
 - (a) Deletion
 - (b) Insertion
 - (c) Sorting
 - (d) All
7. The situation when in a linked list $START = NULL$ is:
 - (a) underflow
 - (b) overflow
 - (c) house full
 - (d) All
8. Which of the following is a two-way list?
 - (a) grounded header list
 - (b) circular header list
 - (c) linked list with header and trailer nodes
 - (d) None of these
9. In which of the following type of linked list, each node contains the "next link field" and "previous link field"?
 - (a) singly linked list
 - (b) circularly single-linked list
 - (c) doubly linked list
 - (d) None of these
10. Which of the following statement is false?
 - (a) Linked list is a dynamic data structure.
 - (b) Data elements in the linked list need not be stored in adjacent space in memory.
 - (c) Pointers store the next data element of a list.
 - (d) Linked list is a collection of the nodes that contains the information part and next pointer.

11. What is the time complexity to count the number of elements in the linked list?
(a) $O(1)$ (b) $O(n)$ (c) $O(\log n)$ (d) $O(n^2)$
12. What would be the time complexity to add an element in the linked list?
(a) $O(1)$ (b) $O(n)$ (c) $O(n^2)$ (d) $O(\log n)$
13. A variant of linked list in which the last node of the list points to the first node of the list is?
(a) Singly linked list (b) Doubly linked list
(c) Circular linked list (d) Multiply linked list
14. In doubly linked lists, traversal can be performed?
(a) Only in the forward direction (b) Only in the reverse direction
(c) In both directions (d) None
15. In the worst case, the number of comparisons need to search a singly linked list of length n for a given element is:
(a) $\log n$ (b) $n/2$ (c) $\log_2 n - 1$ (d) n
16. A linear collection of data element given by the mean of a pointer is called:
(a) Queue (b) Graph (c) Linked List (d) Stack
17. Which of the following is not a type of Linked List?
(a) Singly Linked List (b) Circular Linked List
(c) Hybrid Linked List (d) Doubly Linked List
18. In a circular linked list:
(a) Components are all linked together in some sequential manner.
(b) There is no beginning and no end.
(c) Components are arranged hierarchically.
(d) Forward and backward traversal within the list is permitted.

Answers of MCQs

01 (d)	02 (a)	03 (d)	04 (d)	05 (b)	06 (d)	07 (a)	08 (d)	09 (c)	10 (c)
11 (b)	12 (b)	13 (c)	14 (c)	15 (d)	16 (c)	17 (c)	18 (b)		

- Q.2 What is a linked list? Explain the types of a linked list with examples.
- Q.3 Describe the singly linked list and its representation in memory.
- Q.4 Describe the traversing operation of a singly linked list with an algorithm.
- Q.5 Write an algorithm or code to create a singly linked list and traverse it.
- Q.6 Write an algorithm to display odd numbers stored in a linked list consisting of 15 nodes.
- Q.7 Describe the traversing operation of a singly linked list with an algorithm.
- Q.8 ✓ Write a program to visit a single linked list consisting of five nodes and having a data field of integer type. Calculate the factorial of each value stored in it. Create a new linked list and store the factorials in the newly created linked list. Moreover, display the linked lists.

- Q.9** Write a program to enter records of 10 students using a linked list. Use the following structure of the nodes of the linked list:

```
struct node
{
    char name[15]
    char address[25]
    int marks;
    node *next;
};
```

- Q.10** Describe the doubly linked list and its representation in memory.
- Q.11** Write an algorithm to insert and delete an element in the doubly linked list.
- Q.12** Write an algorithm to visit a double linked list consisting of ten nodes in reverse order.
- ✓ **Q.13** Write a program to insert a new value in a two-way list at a specified location.
- Q.14** Write a program to delete a node from a two-way list at a specified location.
- Q.15** Write a program to modify a value from a single linked list after searching it.
- Q.16** Draw diagrams of a linked list, circular linked list, and doubly linked list with nodes containing the integer values 1, 16, 27, and 92.
- Q.17** Explain some benefits of a doubly-linked list over a singly linked list. Write an algorithm to count the number of nodes in a doubly-linked list.
- Q.18** Write an algorithm to reverse traverse a doubly linked list.
- Q.19** Write an algorithm or code to delete a node with the minimum value from a singly linked list.
- Q.20** Write an algorithm or code that will remove a specified node from a given doubly linked list and insert it at the end of the list.
- Q.21** Write an algorithm or code to transform a circular list into a chain.
- Q.22** Write an algorithm or code to delete all occurrences of x from a given singly linked list.

The data structures that we have discussed in previous chapters are linear data structures. The examples of these linear data structures are array, queue, linked list, and stack. However, there are many applications where linear data structures are not applied (used). In such cases, there is a need for some non-linear data structures such as tree and graph. In this chapter, an introduction to the tree data structure is given. The advanced topics about tree data structure are given in the next chapter i.e. chapter 11.

10.1 TREE

Tree is the most widely used *abstract data type* (ADT) or data structure. It is a non-linear data structure. It is a collection of nodes (or entities) that are organized in a hierarchical structure (without having any cycle). In a tree data structure, every individual entity or element is called a *node*. The nodes are connected by edges (The connection line between two nodes is called an *edge*). Each node contains a value or data. The first node of the tree is called the *root node* and it may contain several links.

Suppose a person's name is Majeed and he has four children, Aslam, Saleem, Haleem, and Naseem.

- ★ Aslam has three children, Aqsa, Hadia and Iqra
- ★ Saleem has four children, Sajid, Aysa, Sadia, and Fatima
- ★ Haleem and Naseem are un-married.

The graphical representation of Majeed's family is shown in figure 10.1.

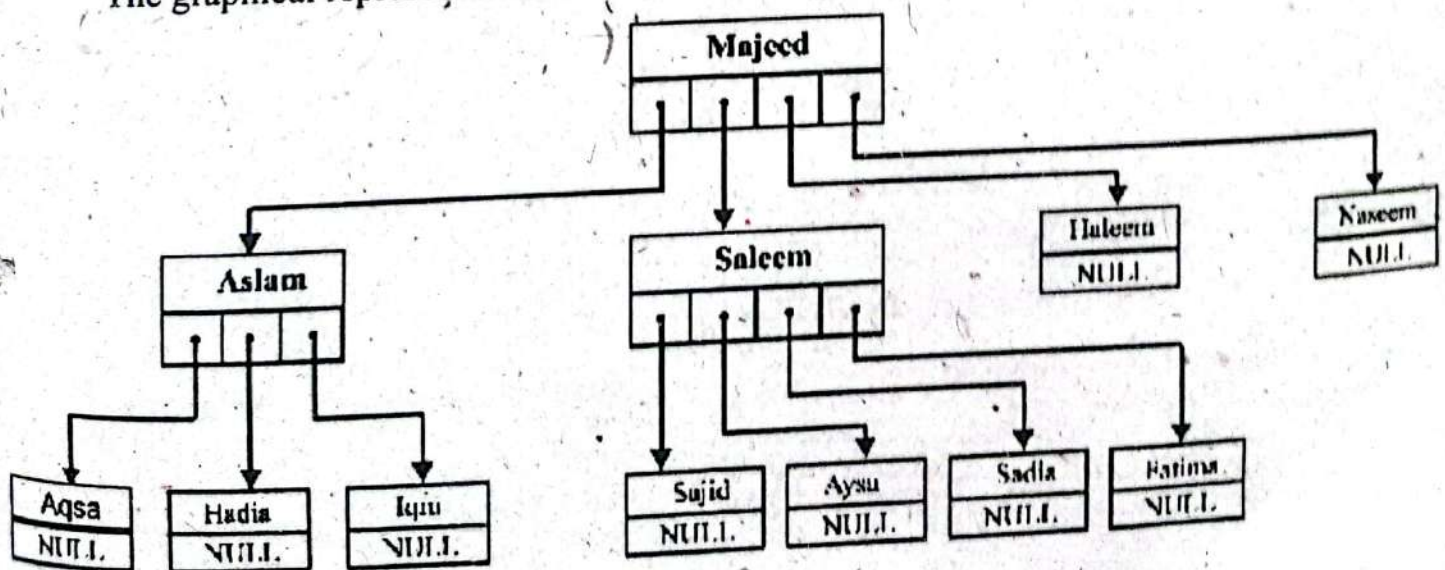


Figure 10.1: Graphical representation of Majeed's family

In this family tree structure, each node contains a name (for data) and has one or more links (pointers) that point to the other nodes. It is a general tree structure.

Tree is a dynamic data structure in which nodes are added and removed dynamically during program execution. The nodes are linked to each other through a chain of pointers. Tree data structure is used in various computer applications.

In a tree, any of the nodes cannot be duplicated. In a linked list, each node has a link that points to the next node (points to the previous node as well – in case of a doubly-linked list). In a tree structure, however, each node may point to several other nodes. Therefore, each node of a tree structure stores actual data and links of other related nodes. Examples of a tree can be a company's organizational chart and a family structure (i.e. family tree).

A tree with no nodes is called a *null tree* or *empty tree*. A non-empty tree consists of a *root node* and possibly many levels of additional nodes that form a hierarchy. If the root node is connected by another node, then the root node is called the *parent node* and the connected node is called a *child node*. A typical tree is shown in the following figure:

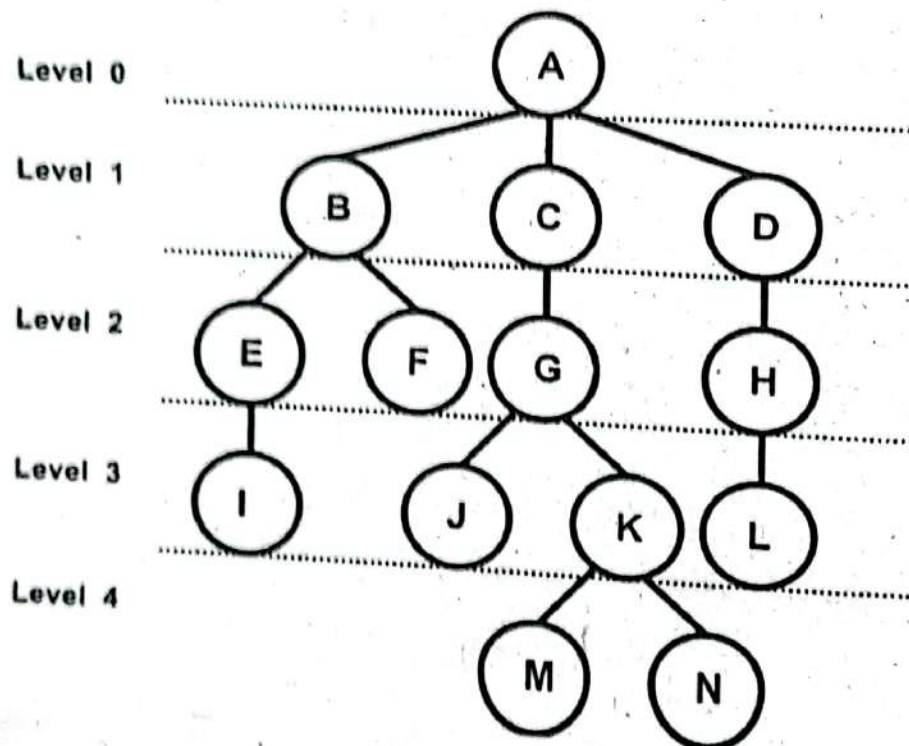


Figure 10.2: Tree

10.1.1 Basic Terminologies Related to Trees

Consider the tree shown in figure 10.2 to understand the basic terminologies related to trees. These are as follows:

1. Node

In a tree data structure, every individual element or entity is called a *node*. It holds data and links to other nodes (child nodes). In figure 10.2, all small circles or bubbles represent the nodes of a tree.

Root Node

The topmost node in the tree that has no parent node is called the *root node*. It represents the beginning of the tree. Element A in figure 10.2 is the root node of the tree.

Every tree must have a root node. We can say that the root node is the origin of a tree data structure. In any tree, there must be only one root node and one path from the root node to any node. A tree can never have multiple root nodes.

Parent Node

A node that has one or more child nodes is called a *parent node*. It is also known as an *ancestor node*. It is always above its child nodes. The node A in figure 10.2 is the root node. It is also the parent node of nodes B, C, and D. Similarly, node B is the parent node of nodes E and F, node C is the parent node of node G, node D is the parent node of node H, and so on.

Note that a root node is the only node in the tree which has no parent. Except for the root node, each node of the tree has one parent. Any node cannot have more than one parent.

Child Node

The node that is directly connected to a parent node is called the *child node*. In figure 10.2, nodes B, C, and D are the child nodes of A. Child nodes are always written one level below the parent.

In a tree, any parent node can have any number of child nodes. All the nodes of a tree except the root node are child nodes.

Siblings

The nodes which have the same parent are called *siblings*. The nodes E and F in figure 10.2 are siblings as both these nodes are child nodes (children) of node B.

Leaf Node

A node that does not have any child node (or successor) is called the *leaf node* or a *leaf*. It is also called a *terminal node* or an *external node*. The nodes I, F, J, M, N, and L in figure 10.2 are terminal nodes or leaves. A tree can have only one root node and many leaf nodes (or leaves).

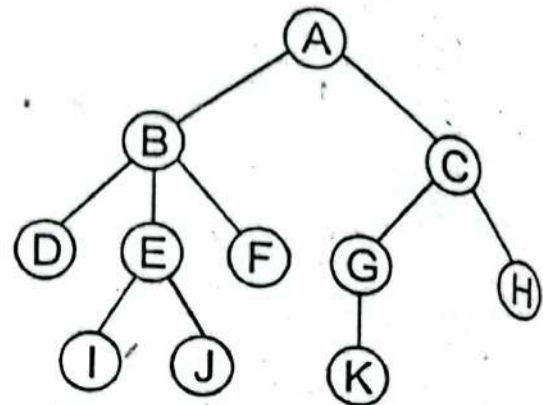
Ancestor & Descendants

An ancestor is a person's forefather such as his/her father, grandfather or grandmother, etc. and so forth. Descendants are the person's children, grandchildren, and so forth. In the following figure, node A is the root of the tree, and node B is the root of its left subtree. Node A is said to be the *father* of node B and node B is said to be the *left son* of node A.

Node 'n1' is an *ancestor* of node 'n2' and node 'n2' is a *descendant* of node 'n1' if node 'n1' is either the father of node 'n2' or the father of some ancestor of node 'n2'. In other words, a parent is an ancestor to its children and its children are descendants. For example, in the following tree:

Ancestors:

- Nodes E, B, and A are ancestors of node J.
- Nodes C and A are ancestors of node H.
- Node A is an ancestor of both nodes J and H.
- Node A is the ancestor of all other nodes in the tree.



Descendants:

- Node H is a descendant of node C.
- Nodes E, I, and J are descendants of node B.
- Every node of the tree other than node A is a descendant of node A.

8. Internal Node

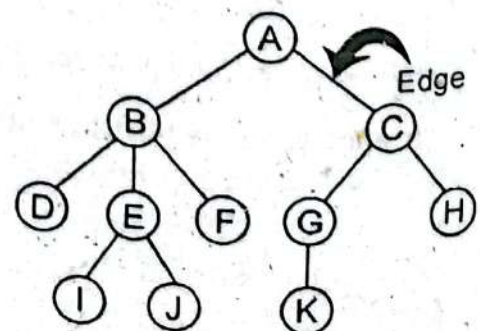
A node that has at least one child (i.e. non-leaf node) is called an *internal node*. It is also called an *interior node*. In other words, all nodes other than leaf nodes are called *internal nodes*. The root node is also said to be an *internal node* if the tree has more than one node. Internal nodes are also called as *non-terminal nodes*.

9. Level of a Node

The level of a node represents the rank of hierarchy in a tree. The root node is said to be at Level 0 and the children of the root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on. In simple words, in a tree, each step from top to bottom is called a *Level* and the Level count starts with '0' and is incremented by one at each level (step). For example in the tree of figure 10.2, the nodes I, J, K, and L are at level 3, and M & N are at level 4.

10. Edge

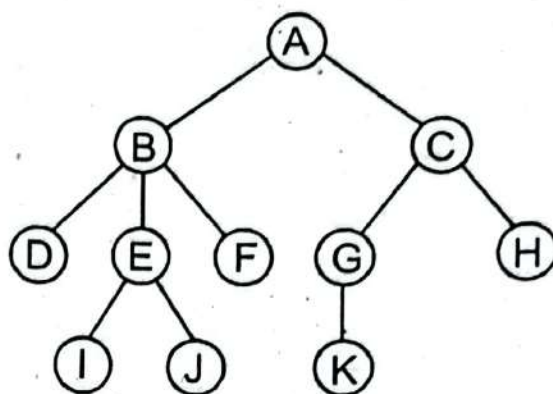
A connection from one node to another node is called an *edge*. In other words, a line drawn from a parent node to its child node is said to be an *edge*. The nodes are connected by edges. In a tree with 'N' number of nodes, there will be a maximum of 'N-1' number of edges.



Path & Length of Path

A *path* is a sequence of nodes and edges between two specific nodes. In the following figure, the path between A and J is A - B - E - J. Similarly, the path between C and K is C - G - K. In a tree, there is exactly one path from the root to each node.

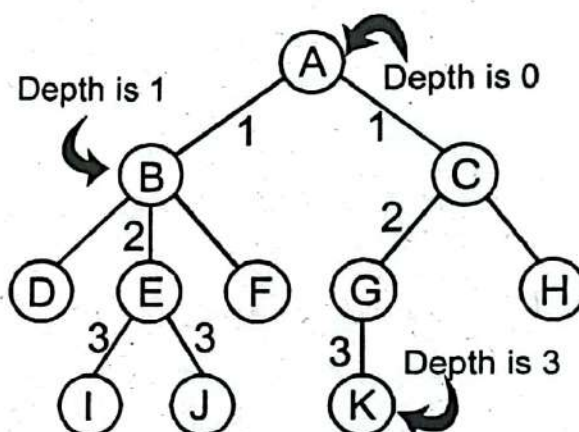
The length of a path is the total number of edges in that path. The length of the path from every node to itself is zero. In the following figure, the path between A and J is A - B - E - J and has length 3.



Depth & Depth of Tree

In a tree data structure, the total number of edges (i.e. length of path) from the root node to a particular node is called the *depth* of that node. The depth of the root node is '0'.

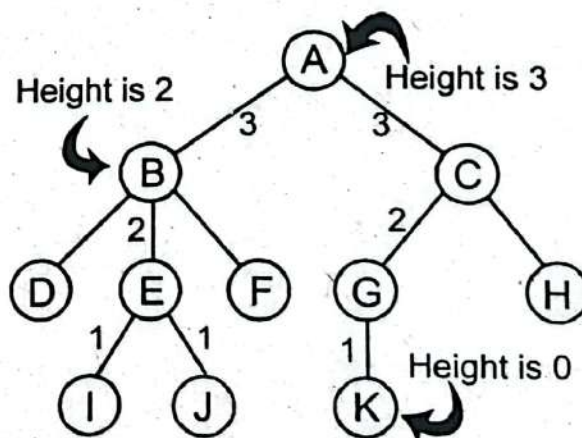
In a tree, the total number of edges from the root node to a leaf node that exists on the longest path is said to be the *depth of the tree*. In simple words, the highest depth of any leaf node in a tree is said to be the *depth of that tree*.



Height & Height of Tree

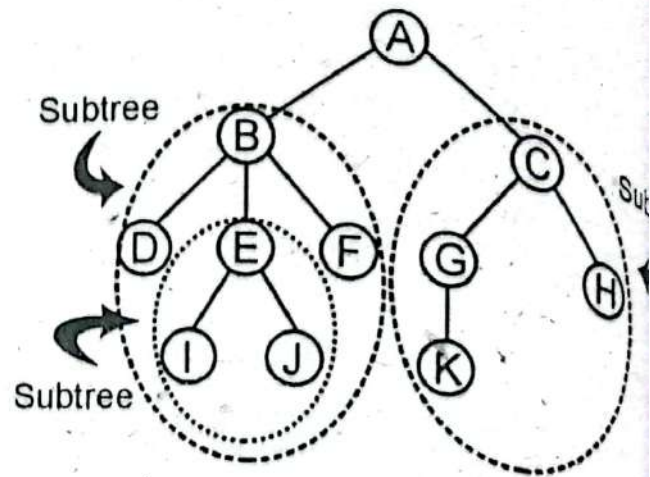
In a tree data structure, the total number of edges on the longest path between a particular node and a leaf is called the *height* of that node. Thus in a tree, the height of all leaf nodes (leaves) is '0'.

The number of edges on the longest path from the root to the leaf is called the *height of the tree*. In other words, the height of a tree is equal to the height of the root. The height of the tree in figure 10.2 is 4.



14. Subtree

The child node of the root node that has its own child nodes is called a *subtree*. The figure shows subtrees and the roots of these subtrees are B, C, and E.

**15. Degree of Node**

The total number of children (child nodes) of a node is called the *degree of that node*. In figure 10.2, the degree of the node E is one and the degree of node K is two.

16. Empty Tree

A tree that has no node is called an *empty tree* or *null tree*. Its height is -1 .

17. Singleton Tree

A tree whose root is its only node is called a *singleton tree*.

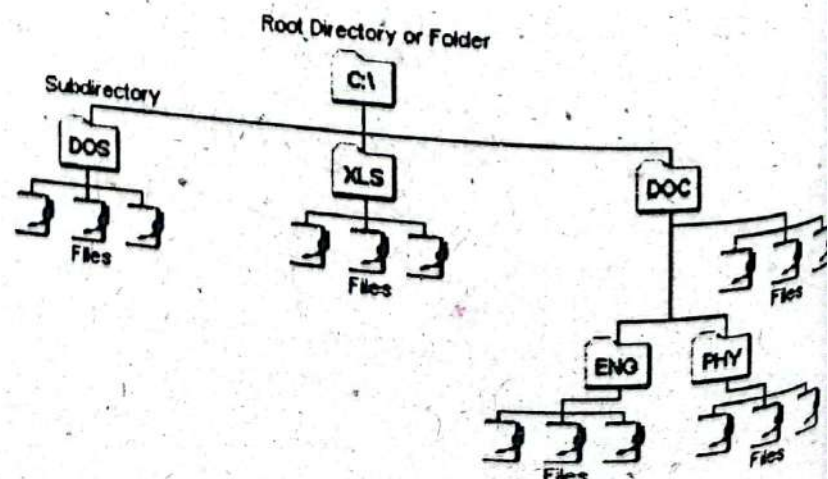
18. General Tree

A tree in which a node may have no child, one child, or any number of children is called a *general tree*.

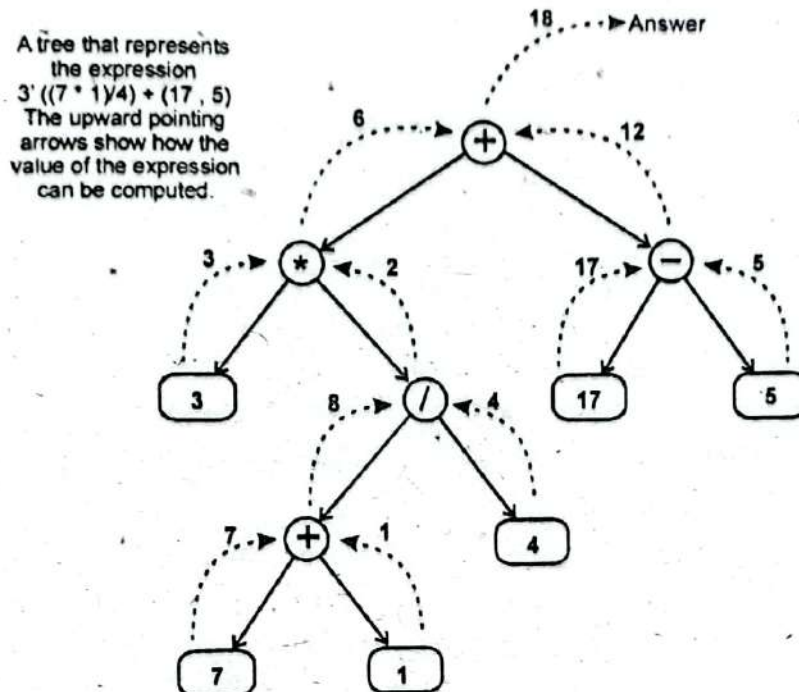
10.1.2 Applications of Tree Data Structure

Tree is a very important data structure. In computer science, the tree data structure is used for many purposes. Some of these are as follows:

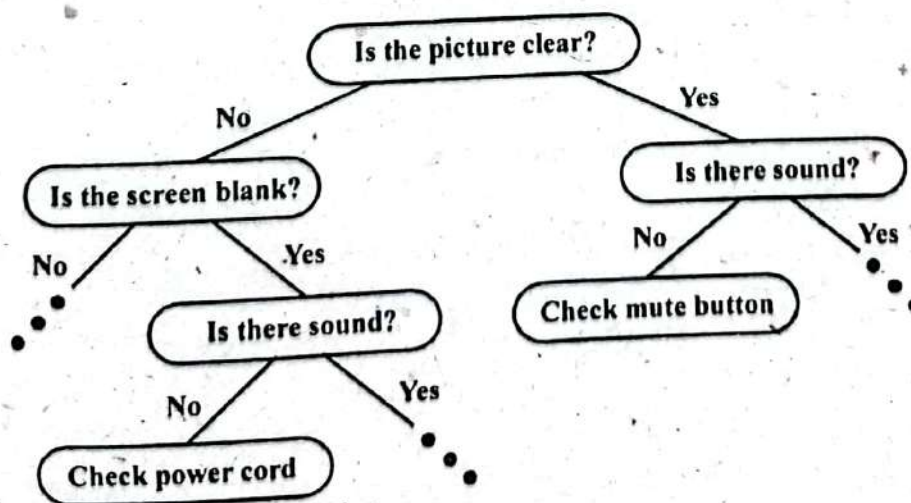
- **File Systems:** It is used to implement hierarchical file systems of several popular operating systems such as UNIX and DOS. In DOS, each '\ ' represents an edge, and in UNIX, it is represented as '/'. Each directory/folder contains subdirectories and files which represent its children (child nodes).



- **Data Storage:** It is used to organize a large volume of data in a linked structure. Data stored in this manner can be accessed and searched more efficiently.
- **Expression Evaluation:** It can be used to evaluate arithmetic expressions. When a parenthesized expression is represented by a tree, each operand becomes a leaf node and each operator becomes an internal node. We can evaluate an expression very easily as shown here.



- **Compiler:** It is used for translating program source code into machine code (i.e. binary code).
- **Making Decision:** A tree data structure can be used in a complicated decision-making process. For example, the following tree structure can be used to find out (making decisions) the fault of a television set.



- **Image Processing:** Tree data structure is suitable for digital image processing for visual effects.
- **Searching:** It provides a quick and efficient way of searching (using tree traversals).

10.2 ✓ BINARY TREE

A tree in which each node can have a maximum of two children (zero, one, or two child nodes) is called a *binary tree*. A binary tree may be empty or non-empty but a general tree cannot be empty. A non-empty binary tree has the following three parts:

Root Node

The node of the binary tree that has no parent is called the *root node*. It is the starting node of any binary tree.

Left Child Node

The node that lies on the left side of the root of a binary tree (or root of a subtree) is called the *left child node*. Node B in figure 10.4 is the left child node of A.

Right Child Node

The node that lies on the right side of the root of a binary tree (or root of a subtree) is called the *right child node*. Node C in figure 10.4 is the right child node of A.

Both the child nodes of the root of a binary tree (or subtree) may be empty or either one of the child nodes may be empty. An example of a binary tree is shown in figure 10.4. In this figure, A is the root of the binary tree, and B & C are left and right children respectively, of the root A. The right child C also has its own left and right children.

Binary tree is a very useful data structure. It is used to keep data that needs to be searched quickly, or that needs to be maintained in sorted order with lots of insertions or deletions.

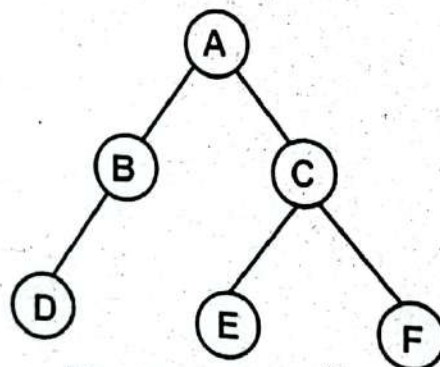


Figure 10.4: Binary Tree

Types of Binary Tree

Following are the major types of binary tree:

- | | |
|---------------------------|-------------------------|
| i) Full binary tree | ii) Perfect binary tree |
| iii) Complete binary tree | iv) Degenerate tree |
| v) Skewed binary tree | vi) Degenerate tree |
| vii) Extended binary tree | |

10.2.1 ✓ Full Binary Tree

A binary tree in which every node other than leaves has two children is called a *full binary tree*. In this type of binary tree, every node has either 0 or 2 children. Full binary tree is used to represent mathematical expressions. The full binary trees are shown in the following figures:

Internal Node

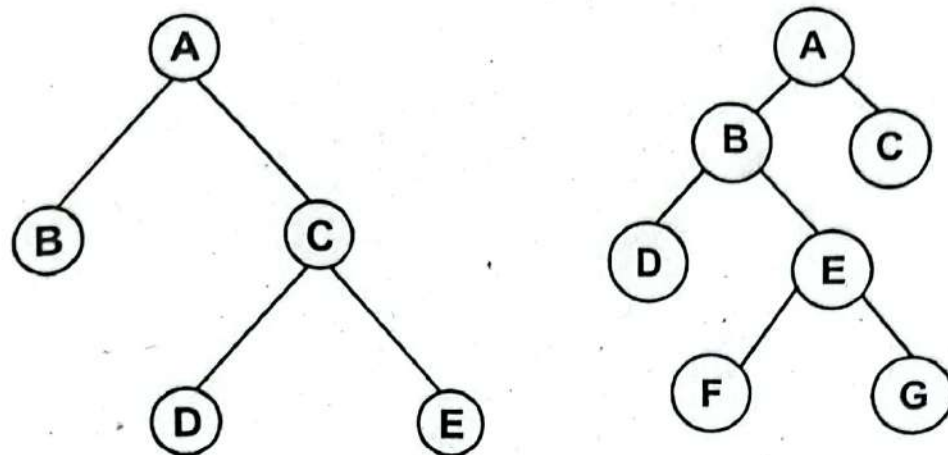


Figure 10.5: Full Binary Trees

0.2.2 Perfect Binary Tree

A binary tree in which all interior or internal nodes have two children and all leaves have the same depth or the same level is called a *perfect binary tree*. The perfect binary trees are shown in the following figures:

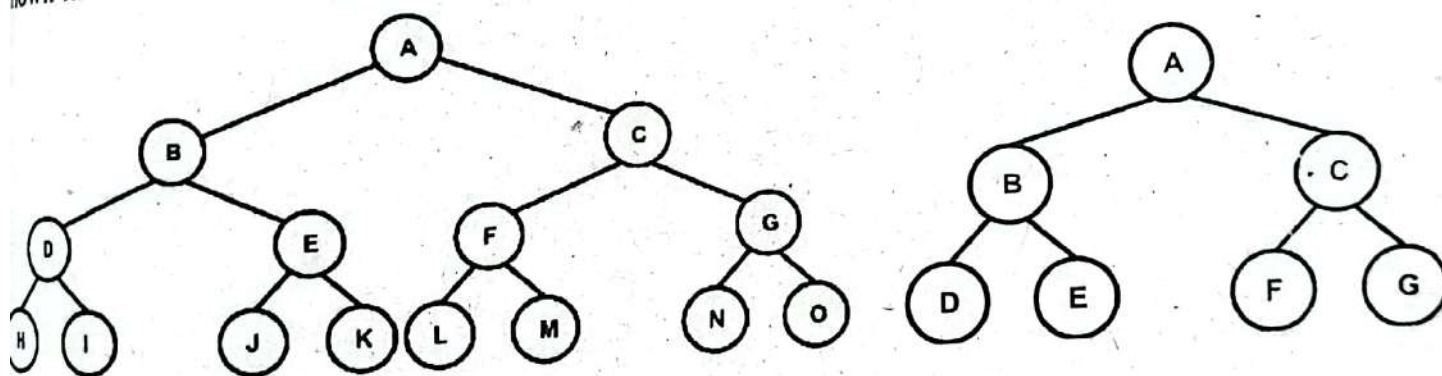
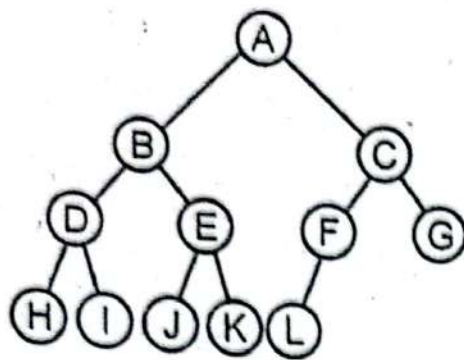


Figure 10.6: Perfect Full Binary Trees

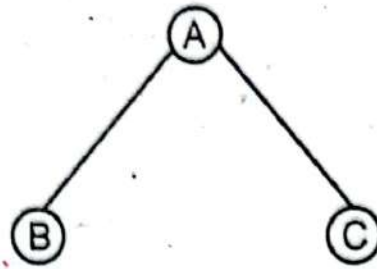
0.2.3 Complete Binary Tree

A binary tree in which every level, except possibly the deepest or last, is completely filled, is called a *complete binary tree*. All nodes in the last level are as far left as possible. In other words, in a complete binary tree, all nodes have two children except the nodes at the lowest two levels. At the lowest level, the nodes have zero children, and at the level above that nodes can have 0, 1, or 2 children. Moreover, the last level must be filled from left to right without leaving any gaps.

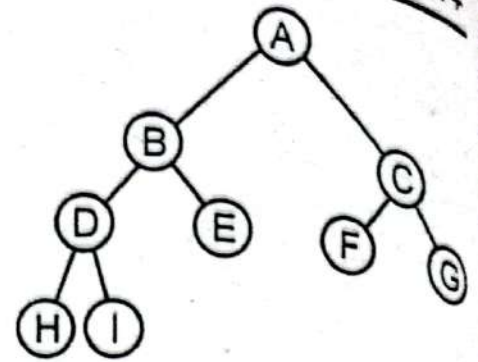
A typical complete binary tree is shown in figure 10.7 (a). A binary tree shown in figure 10.7 (b) is a full binary tree as well as a complete binary tree. It means that a full binary tree can be a complete binary tree and a complete binary tree may also be a full binary tree. But it is not always true. Figure 10.7 (c) shows a complete and full binary tree.



(a) Complete Binary Tree



(b) Full & Complete Binary Tree



(c) Complete & Full Binary Tree

Figure 10.7: Complete Binary Trees

10.2.4 Degenerate Tree

A tree where for each parent node, there is only one associated child node (either left or right), is called a *degenerate tree*. It is also called a *pathological tree*.

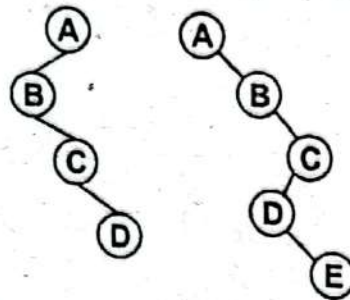
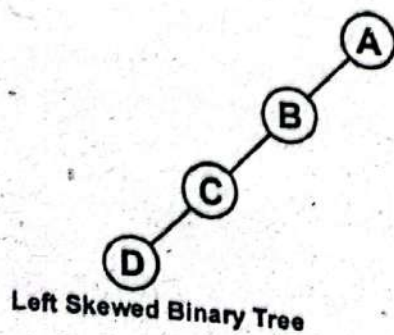


Figure 10.8: Degenerate Trees

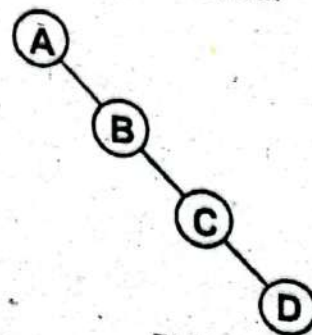
10.2.5 Skewed Binary Tree

A binary tree in which every internal node has only one child is called a *skewed binary tree*. A skewed binary tree is divided into two categories/forms:

- **Left Skewed Binary Tree:** In this form of a skewed binary tree, all internal nodes have the left child without the corresponding right child.
- **Right Skewed Binary Tree:** In this form of a skewed binary tree, all internal nodes have the right child without the corresponding left child.



Left Skewed Binary Tree



Right Skewed Binary Tree

Figure 10.9: Left and Right Skewed Binary Trees

Note that all skewed trees are degenerate or pathological trees, but all pathological trees are not skewed trees.

10.2.6 ✓ Balanced Binary Tree

A binary tree in which the left and right subtrees of every node differ in height by no more than one is called a *balanced binary tree*. It means that a binary tree is said to be a balanced binary tree if the difference between the height of the left subtree and right subtree is maximum one for all the nodes.

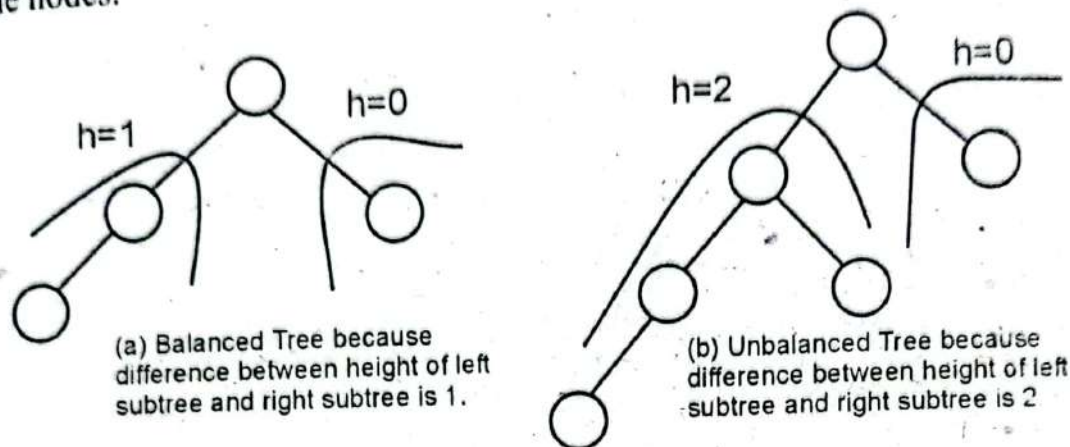


Figure 10.10: Balanced and Unbalanced Binary Trees

10.2.7 ✓ Extended Binary Tree (2-Tree) External Node Add;

A binary tree in which special nodes are added in place of every null subtree is called an *extended binary tree*. It is also called *2-Tree*. The nodes from the original tree are called *internal nodes* and the special nodes are called *external nodes*. The internal node is represented by a circle and the external node is represented by a box. The data can be stored either in the internal nodes or the external nodes.

Every internal node in the extended binary tree has exactly two children and every external node has no children (i.e. external nodes are the leaves of the extended tree). It displays the result which is a *complete binary tree*. Figure 10.11 (a) shows a binary tree before adding special nodes and figure 10.11 (b) extended binary tree.

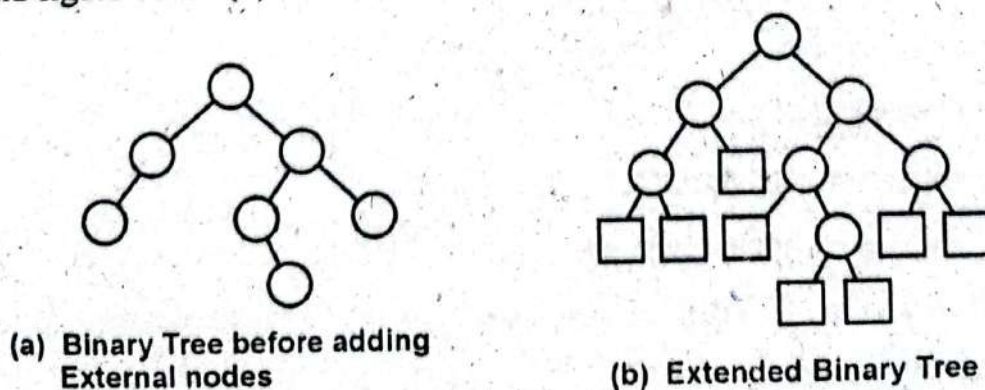


Figure 10.11: Non-Extended and Extended Binary Tree

10.3 ✓ BINARY SEARCH TREE (BST)

A binary tree in which every node contains smaller values in its left subtree and larger values in its right subtree is called a *binary search tree (BST)*. For example, in BST, all the nodes in the left subtree of the root node contain smaller values than the value of the root node, while all the nodes in the right subtree contain larger values than the value of the root node. Similarly, every node of BST satisfies this condition. In figure 10.12, the tree on the left side is a binary search tree, but the tree on the right is not. The tree on the right has a node with item 7 in the left subtree which is greater than the value of the root node.

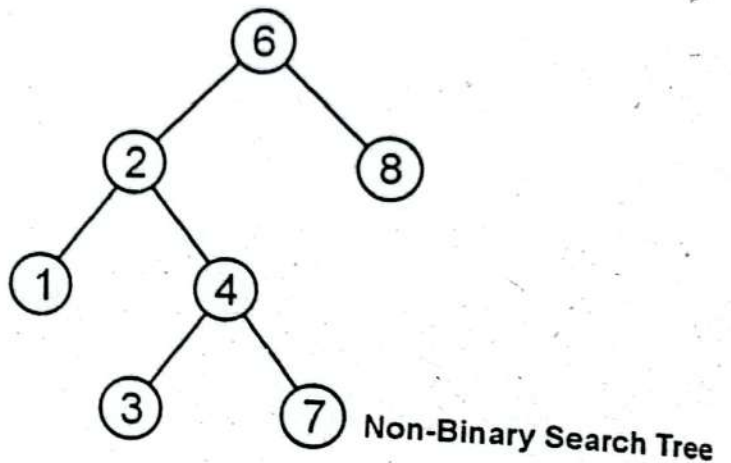
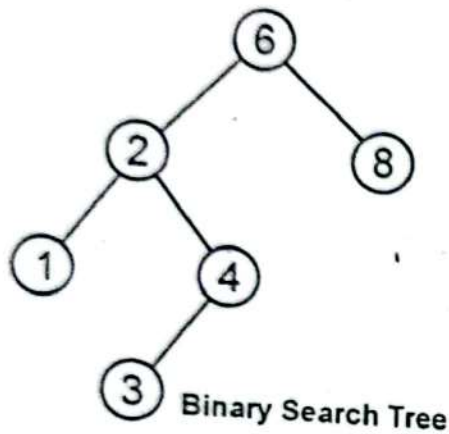
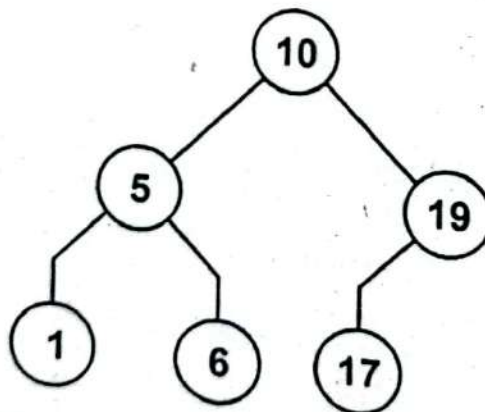


Figure 10.12: Binary search tree and non-binary search tree

Consider the following binary search tree:



In this binary search tree:

- The value of the root node is 10.
- The values in the left subtree are: {5, 1, 6}
- The values in the right subtree are: {19, 17}

Note that all values in the left subtree are less than 10, while all values in the right subtree are greater than 10.

The main advantage of a binary search tree is that it is easy to perform operations like creating a new tree, insertion, searching, and deletion of data.



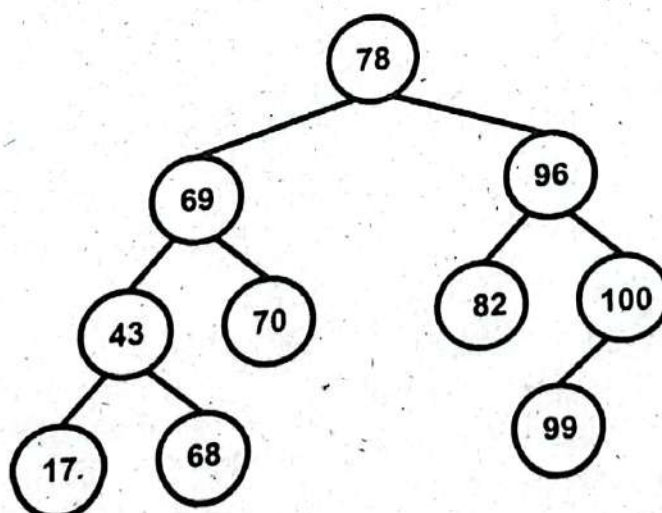
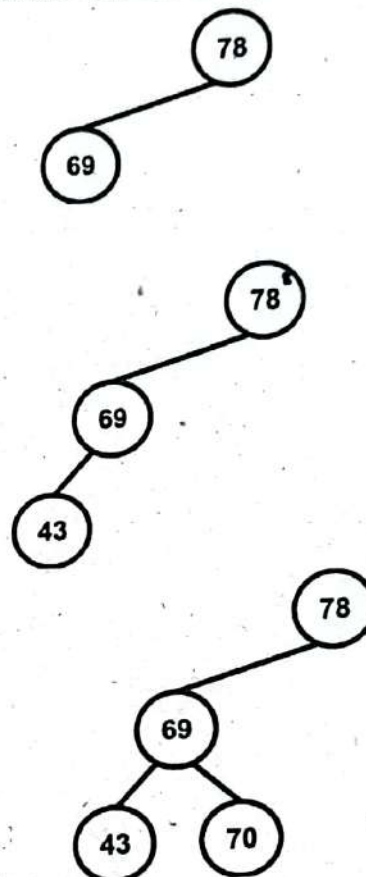
Every binary search tree is a binary tree but the reverse may not be true (i.e. every binary tree may not be a binary search tree).

Constructing Binary Search Tree

Suppose a binary search tree has the following values is to be constructed:
78, 69, 43, 70, 17, 96, 68, 82, 100, 99

The following rules are followed to construct the binary search tree on the given values:

1. Select the first value for the root node.
2. Compare the second value with the root value, if it is smaller than the root value then place it on the left side; otherwise place it on the right side. Since 69 is smaller than 78, place it on the left side.
3. Repeat step-2 for the third value i.e. compare 43 with all the values starting from the root. It is smaller than 78, so go to the left side of the root and compare it with the left child node value. It is also smaller than 69, so place it on the left side of this node.
4. Compare the fourth value i.e. 70 with the root value. It is smaller than the root value. Compare it with the left child node, it is larger than 69. Place it to the right of this node.
5. By following the same procedure, the binary search tree for the other given values is constructed as shown in the figure below:



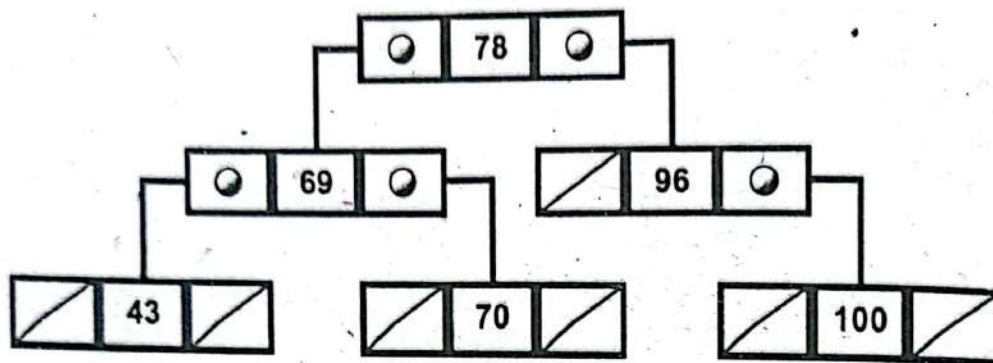
0.3.1 Representation of Binary Search Tree in Memory

Different techniques can be used to represent a binary search tree in the computer memory. The most commonly used techniques are as follows:

- i) Linked storage technique
- ii) Sequential storage technique

10.3.1.1 Linked Storage Technique (Using Linked List)

Linked storage technique is the most commonly used method for representing binary trees inside the computer memory. In this technique, the double linked list is used to represent a binary tree. Each node of the binary tree consists of three fields; one for data field and two for pointer fields. The data field is used to store the value of the node. The pointer fields are used to store the memory addresses of the left child node and right child node. A graphical figure of a binary search tree is shown in the following figure using a linked storage technique.



If any node has no child then pointer fields are assigned with value NULL. The pointer field values for both left and right children of leaf nodes are always NULL.

The node structure to represent a binary search tree in the memory is as follows:

Left Child Address	Data	Right Child Address
--------------------	------	---------------------

struct node

```

{
    int data;
    node *left, *right;
};
  
```

This node structure of the tree is similar to a doubly-linked list. Many of the rules that apply to the linked lists also apply to represent binary trees inside the computer.

10.3.1.2 Sequential Storage Technique (using Array)

The sequential storage technique is very simple to represent the tree inside the computer memory. In this technique, a linear array is used to represent the tree inside the computer memory. This technique is efficient if the structure of the tree does not change frequently during its existence in memory. It is best used to represent full binary trees. However, the array is a static data structure.

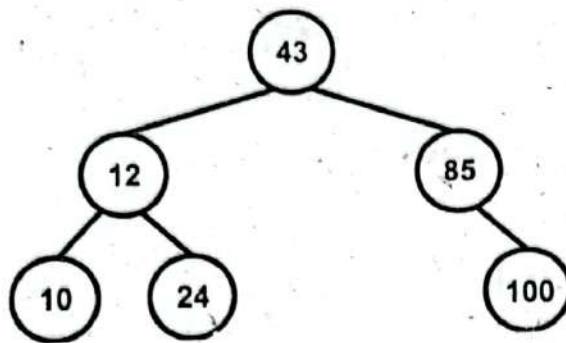
The following rules are used to represent a binary tree using a sequential storage technique (i.e. mapping the values of tree in an array). Suppose the root of a tree is represented by K and the number of nodes by N .

- ★ The value of root 'R' is stored in the first element of the array.
- ★ The value of the left child of the node 'K' is stored in element $2*K$ of the array.
- ★ The value of the right child of the node K is stored in element $2*K+1$ of the array.
- ★ A NULL value indicates that the tree or subtree is empty.

The following figure shows a sequential representation of a binary tree using an array. The tree has the values:

43, 12, 85, 10, 24, 100

The binary tree and its sequential representation are shown below:



Position	0	1	2	3	4	5	6
Elements	43	12	85	10	24		100

Figure: Sequential Representation of Binary Tree

The linked storage technique is recommended to represent complete or full binary trees. If a tree is not full, a lot of elements in the array remain empty. A binary tree of depth 'n' needs an array of approximately 2^{n+1} elements. Thus for a tree that has only 11 nodes and depth 5, the size required to represent the tree is approximately $2^6 = 64$ elements.

Program Code 10.1

// Program that creates a binary search tree and displays the data of all nodes.

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
struct node
```

```
{
```

```
    int n;
```

```
    node *right;
```



```

        node *left;
    };

    class bst
    {
        private:
            node *s,*c,*t;
        public:
            bst(void)
            {
                s = NULL;
            }
            void create(int);
    };

    main()
    {
        bst obj;
        int x[] = {22,33,21,76,88,43,34,11,89,55};
        clrscr();
        for(int i = 0; i<= 9; i++)
            obj.create(x[i]);
    } //end of main()

    // member function to create tree
    void bst::create(int data)
    {
        int col = 40,row = 1;

        // to create root node and assign data to it
        if(s == NULL)
        {
            s = new node;
            s->n = data;
            s->left = s->right = NULL;
            gotoxy(col, row);
            cout<<data;
        }
        else
        {
            node *p;
            c = s;

            // go to proper position to add node
            while(c!= NULL)
            {

```

```

        if(c->n == data)
        {
            cout<<"Data already exists"<<endl;
            getch();
            return;
        }
        if(c->n<data)
        {
            p = c;
            c = c->right;
            row+=2;
            col+=6;
        }
        else
        {
            p = c;
            c = c->left;
            row+=2;
            col-=6;
        }
    } //end of while loop

    // create new node and assign data to it
    t = new node;
    t->n = data;
    t->left = t->right = NULL;
    gotoxy(col, row);
    cout<<data;
    if(p == NULL)
        p = t;
    else if(p->n>data)
        p->left = t;
    else
        p->right = t;
}
} //end of create()

```

Output of the program

```

      22
    21  33
  11    76
      43  88
    34  55  89

```


10.3.2 Operations on Binary Search Trees

Like other data structures, different operations can be performed on the binary search trees too. The most common of these operations are as under:

- | | |
|----------------|----------------|
| i) Search | ii) Traversing |
| iii) Insertion | iv) Deletion |

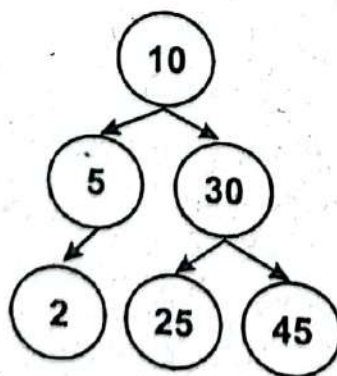
10.3.2.1 Search Operation on BST

Binary search trees can be searched using different methods. Usually, the recursive search method is used to search the binary search trees. It is the easiest way to search a binary tree.

The following steps are taken to search a given value N in the binary search tree using the recursive search method:

1. The search process starts from the root of the tree.
2. If the root value is Null, the Tree is empty and the search process terminates.
3. If the given value N is equal to the value of the root then the search is successful and the searching process terminates.
4. If given value N is smaller than the value of the root, only the left subtree is searched recursively.
5. If given value N is larger than the value of the root, only the right subtree is searched recursively.
6. If the value is found then the search process terminates; otherwise, the whole subtree is searched until its leaves are reached.

Consider the following binary search tree:



Suppose we want to search a node that has value 5. We have to perform the following steps to search this value:

- We will start the search process from the root node.
- The value 5 is smaller than the value of the root node (i.e. $5 < 10$). This value will be searched in the left subtree.
- The value 5 is equal to the value of the root node of the left subtree (i.e. $5 = 5$). The required value is found, so the search process is terminated.

Algorithm – Searching BST

Write an algorithm that searches a node in a binary search tree. Suppose the starting address of the tree is stored in pointer BST. The pointer CUR stores the address of the current node. DATA represents the data field and L & R represent left and right pointers respectively of a node of the tree. Value N represents the value that is to be searched.

1. START
2. INPUT value to search in N
3. IF BST = NULL THEN
 PRINT "Tree is Empty"
 RETURN
 END IF
4. CUR = BST
5. REPEAT STEP-6 TO 8 WHILE CUR != NULL
6. IF N = CUR->DATA THEN
 PRINT "Number is found"
 RETURN
 END IF
7. IF N < CUR->DATA THEN
 CUR = CUR->L
ELSE
 CUR = CUR->R
END IF
8. IF CUR = NULL THEN PRINT "Number not found"
 [End of Step-5 Loop]
9. EXIT

Complexity Analysis of Search Operation on BST

Searching a node in a BST is a binary process. The search algorithm of BST starts the search process from the root node toward the leaf nodes. It takes time proportional to the tree's height. On average, a binary search tree with n nodes has $\Theta(\log n)$ height i.e.

$$n = 1 + 2 + 4 + \dots + 2^{h-1} + 2^h = 2^{h+1} - 1$$

Solving this with respect to 'h', we obtain:

$$h = O(\log n)$$

The binary search tree with n nodes has a minimum of $O(\log n)$ levels, it takes at least $(\log n)$ comparisons to find a particular node. Thus, the average-case time complexity of the search operation of BST is $\Theta(\log n)$. However, in the worst-case, the binary search tree can have $O(n)$ height. We have to traverse all the nodes of height $O(n)$ to find the required node. Thus, the worst-case time complexity of the search operation of BST is $O(n)$.

The space complexity of the search operation of BST is $O(n)$.

Program Code 10.2

//Program to create a binary tree and search a given value in it.

```
#include <iostream.h>
#include <conio.h>

struct node
{
    int data;
    node *right;
    node *left;
};

class bst
{
private:
    node *start, *cur, *parent, *temp;
public:
    bst(void)
    {
        start = NULL;
    }
    void create(int);
    node* search(node*, int);
};

void main(void)
{
    bst obj;
    node *res, *t;
    res = t = NULL;
    int x[] = {22, 33, 21, 76, 88, 43, 34, 11, 89, 55}, val;
    clrscr();
    for(int i = 0; i<=9; i++)
        obj.create(x[i]);
    cout<<"\nEnter value to search :";
    cin>>val;
    res = obj.search(t, val);
    if(res== NULL)
        cout<<"Node not found in this tree :"<<endl;
    else if(res->left== NULL && res->right== NULL)
        cout<<"Node found in this tree. It is a leaf node :"<<endl;
    else if(res->left!= NULL && res->right!= NULL)
        cout<<"Node found in this tree. It is two children node :"<<endl;
    else
        cout<<"Node found in this tree. It is one child node :"<<endl;
    getch();
} //end of main()

// member function to create a binary search tree
void bst::create(int n)
```

```
{  
    int col = 40, row = 1;
```

```
        // to create root node and assign data to it  
    if(start == NULL)
```

```
{
```

```
    start = new node;
```

```
    start->data = n;
```

```
    start->left = start->right = NULL;
```

```
    gotoxy(col, row);
```

```
    cout<<n;
```

```
}
```

```
else
```

```
{
```

```
    cur = start;
```

```
        // go to proper position to add node
```

```
    while(cur!=NULL)
```

```
{
```

```
    if(cur->data == n)
```

```
{
```

```
        cout<<"Data already exists"<<endl;
```

```
        return;
```

```
}
```

```
    if(cur->data < n)
```

```
{
```

```
        parent = cur;
```

```
        cur = cur->right;
```

```
        row += 2;
```

```
        col += 6;
```

```
}
```

```
    else
```

```
{
```

```
        parent = cur;
```

```
        cur = cur->left;
```

```
        row += 2;
```

```
        col -= 6;
```

```
}
```

```
}
```

```
        // create new node and assign data to it
```

```
    temp = new node;
```

```
    temp->data = n;
```

```
    temp->left = temp->right = NULL;
```

```
    gotoxy(col, row);
```

```
    cout<<n;
```

```
    if(parent==NULL)
```

```
        parent = temp;
```

```
    else if(parent->data>n)
```



```

        parent->left = temp;
    else
        parent->right = temp;
    }
} //end of create()

// member function to search a node in the tree
node* bst::search(node* s, int x)
{
    s = start;
    if(s == NULL)
    {
        cout<<"Tree is empty";
        getch();
        return NULL;
    }
    else
    {
        cur = s;
        // search value
        while(cur!= NULL)
        {
            if(cur->data == x)
            {
                return cur;
            }
            if(cur->data < x)
                cur = cur->right;
            else
                cur = cur->left;
        }
    }
    return cur;
} //end of search()

```

10.3.2.2 Traversal Operation

Traversal of a binary search tree is one of the most important operations. It is a process of visiting each node in the tree exactly once. It is also called *visiting*. There are three popular methods used for visiting the binary search tree. These methods are as follows:

1. Inorder Traversal
2. Preorder Traversal
3. Postorder Traversal

Inorder Traversal

1. Using the inorder traversal method, the non-empty binary search tree is visited in the following sequence:

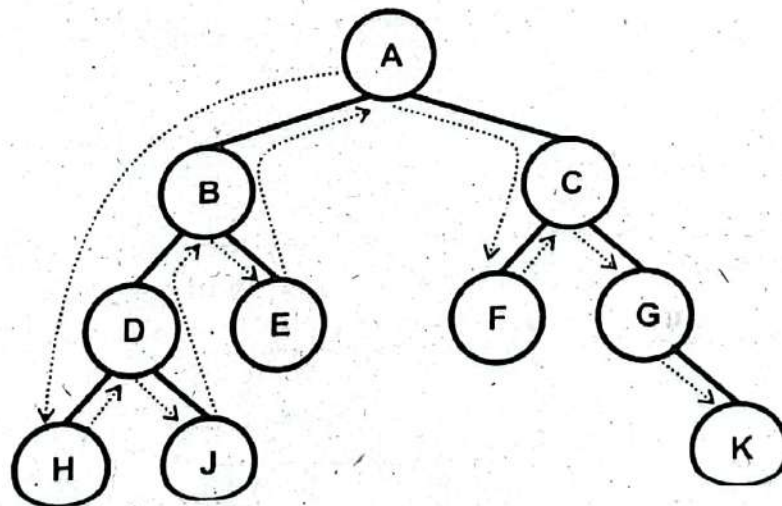
- i) Visit the left sub-tree in inorder
- ii) Visit the root
- iii) Visit the right sub-tree in inorder

When a binary search tree is visited inorder, and the data of the node is accessed, then the data is printed (or displayed) in ascending order.

Usually, a stack is used to implement the inorder traversal method. Moreover, a pointer variable is used to hold the location of the node being visited currently.

Following steps are performed to visit a binary search tree in inorder using stack:

1. Proceed down the left-most path from the root node to the last left node. Push addresses of each left node onto the stack.
2. Pop the top address from the stack. It will be the address of the leftmost node. Visit the node, i.e. access the data of this node to print it.
3. If the node being visited has a right child node, then repeat steps 1 and 2.
4. The above procedure is repeated until all the nodes of the tree are visited.



Inorder traversal of the above binary tree gives the following sequence:

H D J B E A F C G K

Algorithm – Inorder Traversal of Binary Tree

Write an algorithm for the inorder traversal of a binary tree. Suppose the root node address of the binary tree is stored in pointer variable T. Another pointer variable P holds the address of the current node. STK denotes the stack. TOP denotes the top position of stack STK. PUSH function is used to insert a value onto the stack. The POP function is used to get or remove the value from the top of the stack. Assume L and R represent pointer variables to hold addresses of left and right children.

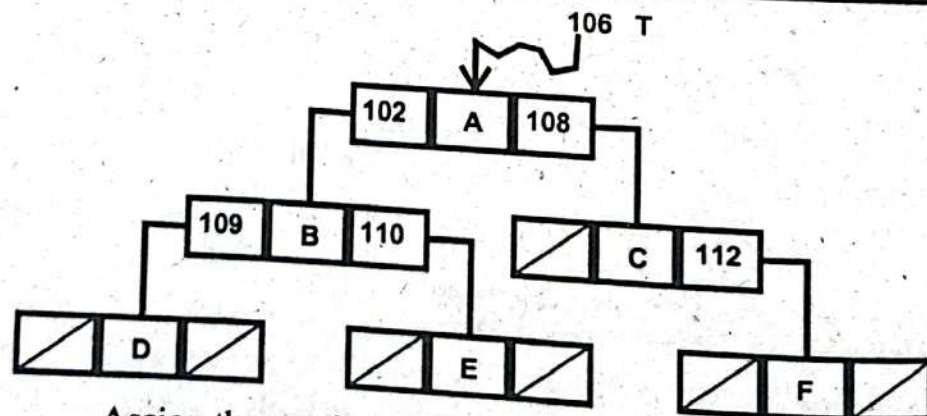
1. [Read starting address of Tree]


```

IF T = NULL THEN
    PRINT "Binary Tree is Empty"
    RETURN
ELSE
    [Initialize the top of stack and assign the address of root to P]
    TOP = 0
    P = T
END IF
2. REPEAT STEP-3 WHILE P != NULL
   [Proceed to the left most node by pushing left node to stack]
3. CALL PUSH(STK, TOP, P)
   P = P->L
   [End of Step-2 loop]
4. REPEAT STEP-5 to 7 WHILE TOP > 0
   [Pop top value of stack STK into P to visit the Tree Inorder]
5. P = POP(STK, TOP)
6. PRINT P->DATA
7. IF P->R != NULL THEN
    P = P->R
    PERFORM STEP-2
END IF
[End of Step-4 Loop]
8. RETURN
    
```

Example:

A Binary Tree is shown below. The values and addresses of its left and right children are also shown. T represents the starting address of the tree and has a value of 106. P represents the current position of the pointer through which the tree is traversed. Explain the steps taken in inorder traversal of the tree using a stack.



Step-1:

Step-2:

Assign the starting address of the tree to P, thus P = 106.
 Loop While P != NULL

Start from the root node, proceed down the left-most path to the last left node while pushing addresses of each left node onto the stack. At the end of the loop, the value in the stack will be 106, 102, 109, and P = NULL

Step-3:

Loop While TOP != NULL

1. The top value of the stack is popped and stored into P, i.e. P = 109. The data of address P, i.e. D is visited.
2. The top value of the stack is popped and stored into P, i.e. P = 102. The data of address P i.e. B is visited.
3. The right address of P is not NULL. Thus loop at step-2 is executed. The values in the stack will be 106 and 110.
4. The top value of the stack is popped and stored into P i.e. P = 110. The data of address P i.e. E is visited.
5. Top value of the stack is popped and stored into P, i.e. P=106. The data of address P, i.e. A is visited. The right address of P is not NULL. Thus step-2 is performed and the value in the stack is 108.
6. The top value of the stack is popped and stored into P, i.e. P=108. The data of address P, i.e. C is visited. The right address of P is not NULL. Thus step-2 is performed and the new value of the stack is 112.
7. The top value of the stack is popped and stored into P, i.e. P= 112. The data of address P, i.e. F is visited.
8. The stack is NULL. The loop condition is satisfied, so the loop terminates.

Program Code 10.3

```
// Program to create and visit a binary search tree using inorder  
// traversal method with the help of stack.
```

```
#include <iostream.h>  
#include <conio.h>
```

```
struct node  
{  
    int n;  
    node *right;  
    node *left;  
};
```

```
class bst  
{
```

```
private:  
    node *s, *c, *t, *stack[10];  
    int top;
```

```
public:  
    bst(void)  
{
```



```

        s = NULL;
        top = -1;
    }

    void create(int);
    void inorder(void);
    void push (node *);
};

void main(void)
{
    bst_obj;
    int x[] = {22,33,21,76,88,43,34,11,89,55};
    clrscr();
    for(int i = 0; i<=9; i++)
        obj.create(x[i]);
    obj.inorder();
    getch();
} //end of main()

// member function to create tree
void bst::create(int data)
{
    int col = 40, row = 1;

    // to create root node and assign data to it
    if(s == NULL)
    {
        s = new node;
        s->n = data;
        s->left = s->right = NULL;
        gotoxy(col, row);
        cout<<data;
    }
    else
    {
        node *p;
        c = s;

        // go to proper position to add node
        while(c->l!=NULL)
        {
            if(c->n == data)
            {
                cout<<"Data already exists"<<endl;
                return;
            }
            if(c->n < data)

```

```

        {
            p = c;
            c = c->right;
            row += 2;
            col += 6;
        }
        else
        {
            p = c;
            c = c->left;
            row += 2;
            col -= 6;
        }
    }

    // create new node and assign data to it
    t = new node;
    t->n = data;
    t->left = NULL;
    t->right = NULL;
    gotoxy(col, row);
    cout << data;
    if (p == NULL)
        p = t;
    else if (p->n > data)
        p->left = t;
    else
        p->right = t;
}
} //end of create()

```

```

// definition of inorder function
void bst::inorder(void)
{
    // read starting address of tree
    if (s == NULL)
    {
        cout << "Tree is empty\n";
        return;
    }
    else
    {
        cout << endl << endl;
        cout << "Traversal in inorder\n";
        c = s;
        push(c);
        cout << endl;
    }
}

```



```

        while(top>=0)
        {
            c = stack[top];
            top--;
            cout<<c->n<<"\t";
            if(c->right!= NULL)
                push(c->right);
        }
    } //end of inorder()

    // definition of push function
    void bst::push(node *cc)
    {
        while(cc!=NULL)
        {
            top++;
            stack[top] = cc;
            cc = cc->left;
        }
    } //end of push ()

```

Output of the program

```

          22
        21  33
       11   76
          43  88
         34  55  89
Traversal in inorder
11  21  22  33  34  43  55  76  88  89

```

Program Code 10.4

// Program to create and visit the binary search tree using inorder recursively

```

#include <iostream.h>
#include <conio.h>

```

```

struct node
{
    int data;
    node *left;
    node *right;
};

```

```

class tree
{

```

```
private:
    node *start, *cur, *temp;
    int top;

public:
    tree()
    {
        start = NULL;
    }
    void create(int);
    void inorder(void);
    void inord(node *s);
};

void main(void)
{
    tree obj;
    int val, p;
    clrscr();
    cout<<"Enter ten values \n";
    for(int i = 1; i<=10; i++)
    {
        cin>>val;
        obj.create(val);
    }
    obj.inorder();
    getch();
} //end of main()

//member function to call the recursive function "inord"
void tree::inorder()
{
    inord(start);
} //end of inorder()

// definition of inord recursive function
void tree::inord(node *s)
{
    if(s!=NULL)
    {
        inord(s->left);
        cout<<s->data<<"\t";
        inord(s->right);
    }
} //end of inord()
```


//member function to create and assign data into tree nodes

void tree::create(int x)

```
{
    if(start==NULL)
    {
        //create root node and assign data
        start = new node;
        start->data = x;
        start->left = NULL;
        start->right = NULL;
    }
    else
    {
        node *parent;
        cur = start;

        //search value in tree and go to proper position to insert
        while(cur!=NULL)
        {
            if(cur->data == x)
            {
                cout<<"Data already exist/n";
                return;
            }
            if(x > cur->data)
            {
                parent = cur;
                cur = cur->right;
            }
            else
            {
                parent = cur;
                cur = cur->left;
            }
        }
        // end of while loop

        // create and insert new node
        temp = new node;
        temp->data = x;
        temp->left = NULL;
        temp->right = NULL;
        if(parent == NULL)
            parent = temp;
        else if(parent->data > x)
            parent->left = temp;
    }
}
```

else

parent->right = temp;

```

    }
} //end of create()

```

Preorder Traversal

Through the preorder traversal method, a non-empty binary search tree is visited in the following order;

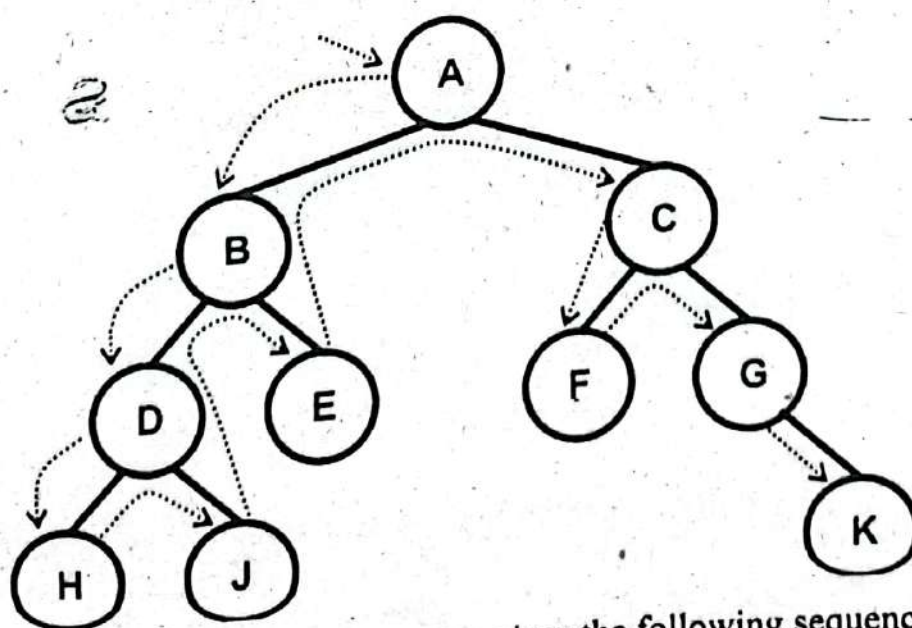
- (i) Visit the root.
- (ii) Visit the left subtree in preorder.
- (iii) Visit the right sub-tree in preorder.

Usually, a stack is used to implement the preorder traversal method. Moreover, a pointer variable is used to hold the location of the node being visited currently.

To visit a binary search tree in preorder using stack, follow these steps:

1. Root node is visited. The address of its right child node (if any) is pushed onto the stack.
2. The left sub-tree of the node is visited. The address of its right child node (if any) is pushed onto the stack.
3. After visiting the last leftmost child node, the top value from the stack is popped and visited in preorder sequence as explained in steps 1 and 2.

Consider the following binary tree:



Preorder traversal of the above binary tree gives the following sequence:

A B D H J E C F G K

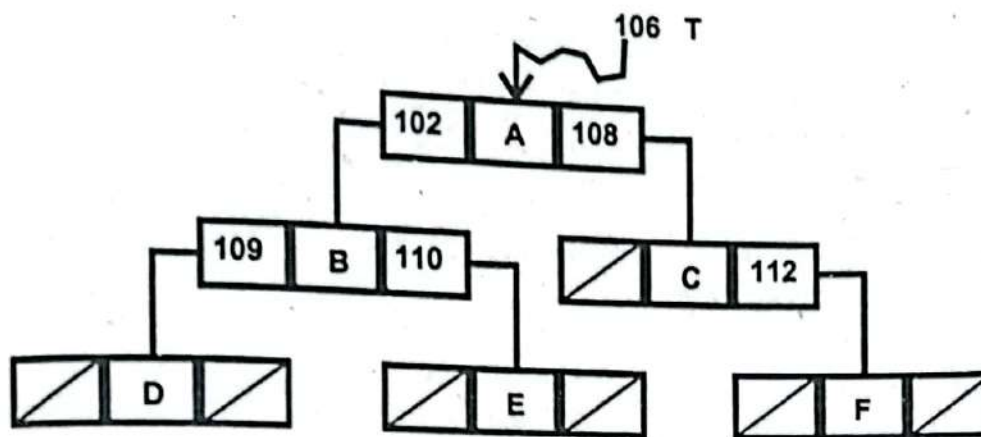
Algorithm – Preorder Traversal of Binary Tree

Write an algorithm for the preorder traversal of the binary search tree. Take a binary tree having its root node address stored in pointer variable T and pointer variable P holding the address of the current node during processing. STK denotes the stack and TOP denotes the top position of the stack STK. The function PUSH is used to insert the value in the stack STK and the function POP is used to remove the value from the top of the stack. Assume L and R represent pointer variables that hold addresses of left & right children.

1. [Read starting address of BST]
 IF T = NULL THEN
 PRINT "Binary Tree is Empty"
 RETURN
 ELSE
 [place pointer to root of Tree on top of stack STK]
 TOP = 0
 CALL PUSH(STK, TOP, T)
 END IF
2. REPEAT STEP-3 TO 4 WHILE TOP > 0
3. [Pop top value of stack STK into P to visit Tree in preorder]
 P = POP(STK, TOP)
4. [Visit the left most nodes of the tree or subtree and assign addresses of each right node to top of the stack]
 REPEAT WHILE P->R != NULL
 PRINT P->DATA [Print data of node]
 IF P->R != NULL THEN
 CALL PUSH(STK, TOP, P->R)
 END IF
 P = P->L [Point to left branch]
 [End of Step-4 Loop]
 [End of Step-2 Loop]
5. RETURN

Example:

Explain the preorder traversal method using a stack of the binary tree shown in the following figure. The addresses of its left & right children are given along with node values. T represents the starting address of the tree with value 106. Assume P represents the current position of the pointer through which the tree is visited.



Step-1: Starting Address of the Tree (i.e. 106) is pushed onto the stack.
The top value of stack = 106

Step-2: Execute Outer Loop WHILE TOP != NULL

First Iteration of Outer Loop

The top value of the stack is popped and the value is assigned to P. Thus
P = 106.

Step-3: Inner Loop While P != NULL

First Iteration of Inner Loop

- 1- Data of address P, i.e. A is visited.
- 2- The right address of P, i.e. 108 is pushed onto the stack. The top value of the stack is 108.
- 3- The left address of P is assigned to P and the new value of P is 102. The control goes to step-3. The value of P is not NULL and the loop iterates once again.

Second iteration of Inner Loop

- 1- Data of address P i.e. B is visited.
- 2- The right address of P, i.e. 110 is pushed onto the stack. The values in the stack are 108 and 110.
- 3- The left address of P is assigned to P and the new value of P is 109. The control goes to step-3. The value of P is not NULL and the loop iterates once again.

Third iteration of Inner Loop

- 1- The data of address P, i.e. D is visited.
- 2- The right address of P is NULL. It is not pushed onto the stack. The values in the stack remain 108 and 110.

- 3- The left address of P is assigned to P and the new value of P is NULL. The condition of the loop is satisfied. The loop terminates.

Second Iteration of Outer Loop

The control shifts back to step-2. As the value in the stack is not NULL, the loop continues and the value of the stack, i.e. 110 is popped and assigned to P. Loop at step-3 is again executed as:

First Iteration of Inner Loop

- 1- The data of address P, i.e. E is visited.
- 2- The right address of P is NULL. It is not pushed onto the stack. The value of the stack remains 108.
- 3- The left address of P is assigned to P and the new value of P is NULL. The condition of the loop is satisfied. The loop terminates.

Third Iteration of Outer Loop

The control shifts back to step-2. As the value in the stack is not NULL, the loop continues and the value of the stack, i.e. 108 is popped and assigned to P. Loop at step-3 is again executed as:

First Iteration of Inner Loop

1. The data of address P, i.e. C is visited.
2. The right address of P i.e. 112 is pushed onto the stack. The top value of the stack remains 112.
3. The left address of P is assigned to P and the new value of P is NULL. The condition of the loop is satisfied. The loop terminates.

Fourth Iteration of Outer Loop

The control shifts back to step-2. As the value of the stack is not NULL, the loop continues and the value of the stack, i.e. 112 is popped and assigned to P. Loop at step-3 is again executed as:

First Iteration of Inner Loop

1. The data of address P, i.e. F is visited.
2. The right address of P, i.e. NULL is pushed onto the stack. The top value of the stack is NULL.
3. The left address of P is assigned to P and the new value of P is NULL. The condition of the loop is satisfied. The loop terminates.

The control shifts back to step-2. As the value of the stack is NULL, the loop condition is satisfied. The loop terminates.

Program Code 10.5

// Program to create and visit the binary search tree using preorder traversal method with the help of stack.

```
#include <iostream.h>
#include <conio.h>
```

```
struct node
{
```

```
    int n;
    node *right;
    node *left;
```

```
};
```

```
class bst
```

```
{
```

```
    private:
```

```
        node *s, *c, *t, *stack[10];
        int top;
```

```
    public:
```

```
        bst(void)
```

```
        {
```

```
            s = NULL;
            top = -1;
```

```
        }
```

```
        void create(int);
```

```
        void preorder(void);
```

```
};
```

```
void main(void)
```

```
{
```

```
    bst obj;
```

```
    int x[] = {22, 33, 21, 76, 88, 43, 34, 11, 89, 55};
```

```
    clrscr();
```

```
    for(int i = 0; i <= 9; i++)
        obj.create(x[i]);
```

```
    obj.preorder();
```

```
    getch();
```

```
} //end of main()
```

```
    // member function to create tree
```

```
void bst::create(int data)
```

```
{
```

```
    int col = 40, row = 1;
```

```
    // to create root node and assign data to it
```

```
    if(s == NULL)
```

```
    {
```

```
        s = new node;
```



```

        s->n = data;
        s->left = s->right = NULL;
        gotoxy(col,row);
        cout<<data;
    }
    else
    {
        node *p;
        c = s;

        // go to proper position to add node
        while(c!=NULL)
        {
            if(c->n == data)
            {
                cout<<"Data already exists"<<endl;
                return;
            }
            if(c->n < data)
            {
                p = c;
                c = c->right;
                row+= 2;
                col+= 6;
            }
            else
            {
                p = c;
                c = c->left;
                row+= 2;
                col-= 6;
            }
        }

        // create new node and assign data to it
        t = new node;
        t->n = data;
        t->left = NULL;
        t->right = NULL;
        gotoxy(col, row);
        cout<<data;
        if(p==NULL)
            p = t;
        else if(p->n>data)
            p->left = t;
        else
            p->right = t;
    }
}

```

} //end of create()

void bst::preorder(void)

```
{
    // read starting address of tree
    if(s == NULL)
    {
        cout<<"Tree is empty\n";
        return;
    }
    else
    {
        cout<<endl<<endl;
        cout<<"Traversal in preorder\n\n";
        top++;
        stack[top] = s;
        while(top>=0)
        {
            c = stack[top];
            top--;
            while(c!=NULL)
            {
                cout<<c->n<<"\t";
                if(c->right!=NULL)
                {
                    top++;
                    stack[top] = c->right;
                }
                c = c->left;
            }
        }
    }
} //end of pre-order()
```

Output of the program

```

      22
     21 33
    11 76
      43 88
     34 55 89

```

Traversal in preorder
22 21 11 33 76 43 34 55 88 89

Program Code 10.6

// Program to create and visit binary search tree preorder recursively
#include <iostream.h>


```
#include <conio.h>
```

```
struct node
```

```
{
```

```
    int data;
```

```
    node *left;
```

```
    node *right;
```

```
};
```

```
class tree
```

```
{
```

```
    private:
```

```
        node *start, *cur, *temp;
```

```
        int top;
```

```
    public:
```

```
    tree() { start = NULL; }
```

```
    void create(int);
```

```
    void preorder(void);
```

```
    void preord(node *s);
```

```
};
```

```
void main(void)
```

```
{
```

```
    tree obj;
```

```
    int val, p;
```

```
    clrscr();
```

```
    cout<<"Enter ten values\n";
```

```
    for(int i = 1; i<=10; i++)
```

```
    {
```

```
        cin>>val;
```

```
        obj.create(val);
```

```
    }
```

```
    obj.preorder();
```

```
    getch();
```

```
} //end of main()
```

```
//member function to call the recursive function preord
```

```
void tree::preorder()
```

```
{
```

```
    preord(start);
```

```
} //end of preorder()
```

```
//preord recursive function to visit tree in preorder traversal
```

```
void tree::preord(node *s)
```

```
{
```

```
    if(s!=NULL)
```

```
    {
```

```
        cout<<s->data<<"\t";
        preord(s->left);
        preord(s->right);
    }
} //end of preord()

//member function to create and assign data into tree nodes
void tree::create(int x)
{
    if(start == NULL)
    {
        //create root node and assign data
        start = new node;
        start->data = x;
        start->left = start->right = NULL;
    }
    else
    {
        node *parent;
        cur = start;

        // search value in tree and go to proper position to insert
        while(cur!=NULL)
        {
            if(cur->data == x)
            {
                cout<<"Data already exist";
                return;
            }
            if(cur->data < x)
            {
                parent = cur;
                cur = cur->right;
            }
            else
            {
                parent = cur;
                cur = cur->left;
            }
        }

        // create and insert new nodes
        temp = new node;
        temp->data = x;
        temp->left = NULL;
        temp->right = NULL;
        if(parent == NULL)
            parent = temp;
```



```

else if(parent->data > x)
    parent->left = temp;
else
    parent->right = temp;
}
} //end of create()

```

3. Postorder Traversal

Using the postorder traversal method, the non-empty binary search tree is visited as follows;

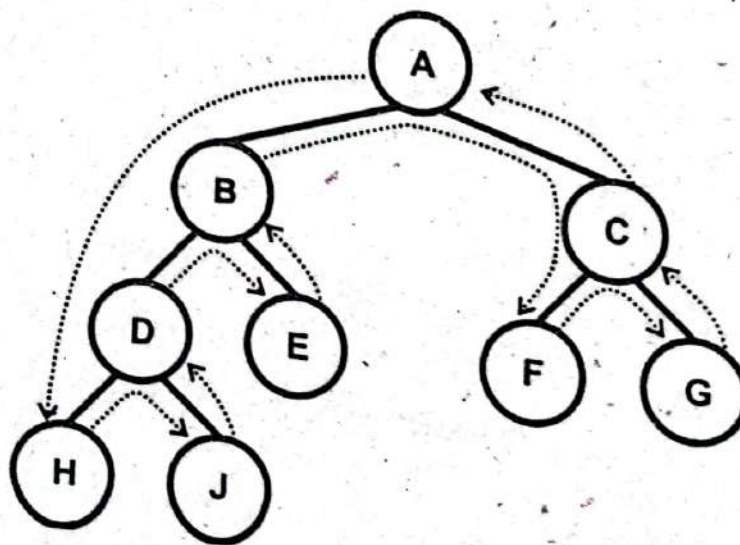
- (i) Visit the left subtree in postorder.
- (ii) Visit the right sub-tree in postorder.
- (iii) Visit the root.

Usually, a stack is used to implement the postorder traversal method. Moreover, a pointer variable is used to hold the location of the node being visited currently.

To visit a binary search tree in a postorder using stack, follow these steps:

1. From the root node, move toward the left node and push the address of each node onto the stack. Continue this process until the last leftmost node is reached.
2. When the last left node is reached, pop the top address from the stack and visit the node. If the sibling right node exists then proceed to the right node and repeat step-1.
3. Visit the root node at the end.

Consider the following binary tree:



Postorder traversal of the tree gives the following sequence:

H J D E B F G C A

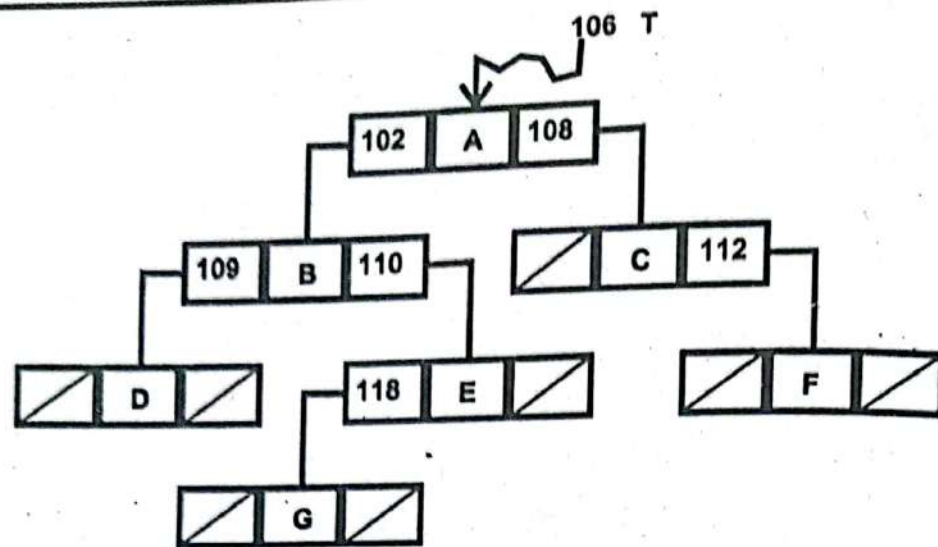
Algorithm – Postorder Traversal of Binary Tree

Write an algorithm for the postorder traversal of a binary tree. Suppose the root node address of the binary tree is stored in pointer variable T. Another pointer variable P holds the address of the current node during processing. STK denotes the stack. TOP denotes the top position of stack STK. PUSH function is used to insert a value onto the stack. The POP function is used to get or remove the value from the top of the stack. Assume L and R represent pointer variables to hold the address of the left & right children.

1. [Read starting address of Tree]
 IF T = NULL THEN
 PRINT "Binary Tree is Empty"
 RETURN
 ELSE [Initialize top of stack and assign root address to P]
 TOP = 0
 P = T
 END IF
2. [Proceed to left most node by pushing left node onto stack]
 REPEAT STEP-3 to 4 WHILE P ≠ NULL
3. CALL PUSH(STK, TOP, P)
4. IF P->L = NULL THEN
 P = P->R
 ELSE
 P = P->L
 END IF
 [End of step-2 loop]
5. [Pop top value of stack into P to visit the tree in postorder sequence]
 REPEAT STEPS-6 to 8 WHILE TOP > 0
6. P = POP(STK, TOP)
7. PRINT P->Data
8. IF P->R ≠ NULL THEN
 P = P->R
 PERFORM STEP-2
 END IF
 [End of step-5 Loop]
9. RETURN

Example:

A Binary Tree is given below. The values and addresses of its left & right children are also shown. T represents the starting address of the tree with value 106. P represents the current position of the pointer through which the tree is traversed. Explain the steps to traverse the tree in postorder sequence using a stack.



Step-1: Assign the starting address of the tree to P i.e. $P = 106$.

Step-2: Loop While $P \neq \text{NULL}$

Starting from the root node, proceed down the leftmost path to the last left node while pushing addresses of each left node onto the stack. At the end of the loop, the values in the stack will be:

106, 102, 109 and $P = \text{NULL}$

Step-3: Loop While $\text{TOP} \neq \text{NULL}$

1. The top value of the stack is popped and stored into P, i.e. $P = 109$.
The data of address P, i.e. D is visited.
The right address of P is not NULL. Thus step-2 is executed and the values in the stack at the end of the loop will be: 106, 102, 110, and 118.
2. The top value of the stack is popped and stored into P, i.e. $P = 118$.
The data of address P, i.e. G is visited.
3. The top value of the stack is popped and stored into P, i.e. $P = 110$.
The data of address P, i.e. E is visited.
4. The top value of the stack is popped and stored into P, i.e. $P = 102$.
The data of address P i.e. B is visited.
The right address of P is not NULL. Thus step-2 is performed and new values in the stack are 106, 108, and 112.
5. The top value of the stack is popped and stored into P, i.e. $P = 112$.
The data of address P, i.e. F is visited.
6. The top value of the stack is popped and stored into P, i.e. $P = 108$.
The data of address P, i.e. C is visited.
7. The top value of the stack is popped and stored into P, i.e. $P = 106$.

The data of address P, i.e. A is visited at the end.

The stack is NULL. The loop condition is satisfied and the loop terminates.

Program Code 10.7

// Program to create and visit binary search tree postorder recursively

```
#include <iostream.h>
#include <conio.h>
```

```
struct node
{
    int data;
    node *left;
    node *right;
};
```

```
class tree
{
private:
    node *start, *cur, *temp;
public:
    tree() { start = NULL; }
    void create(int);
    void postorder(void);
    void post(node *s);
};
```

```
void main(void)
{
    tree obj;
    int val, p;
    clrscr();
    cout<<"Enter ten values \n";
    for(int i = 1; i<=10; i++)
    {
        cin>>val;
        obj.create(val);
    }
    obj.postorder();
    getch();
} //end of main()
```

```
// postorder recursive function to call recursive function
void tree::postorder()
{
```



```

        post(start);
    } //end of postorder()

```

```

    //member function to visit tree recursively
    void tree::post(node *s)
    {

```

```

        if(s!=NULL)
        {
            post(s->left);
            post(s->right);
            cout<<s->data<<"\n";
        }
    }

```

```

} //end of post()

```

```

    //member function to create and assign data into tree nodes
    void tree::create(int x)
    {

```

```

        if(start == NULL)
        {
            //create root node and assign data
            start = new node;
            start->data = x;
            start->left = start->right = NULL;
        }
    }

```

```

    else
    {

```

```

        node *parent;
        cur = start;

```

```

        //search value in tree and go to proper position to insert value
        while(cur!=NULL)
        {

```

```

            if(cur->data == x)
            {

```

```

                cout<<"Data already exist\n";
                return;
            }

```

```

            if(x > cur->data)
            {

```

```

                parent = cur;
                cur = cur->right;
            }

```

```

            else
            {

```

```

                parent = cur;
                cur = cur->left;
            }
        }
    }

```

```

    } // end of while loop

    // create and insert new node
    temp = new node;
    temp->data = x;
    temp->left = temp->right = NULL;
    if(parent == NULL)
        parent = temp;
    else if(parent->data > x)
        parent->left = temp;
    else
        parent->right = temp;
}

} //end of create()

```

Complexity Analysis of Traversal Algorithms of BST

Each traversal algorithm of BST requires constant work at each node (not including recursive calls). Traversing every node of BST with n nodes requires $O(n)$ times. Thus, the worst-case time complexity of each traversal algorithm is $O(n)$. However, the average -case time complexity of each traversal algorithm is $\Theta(\log n)$ for traversing a BST with n nodes having $O(\log n)$ height.

The space complexity of each traversal algorithm is $O(n)$.

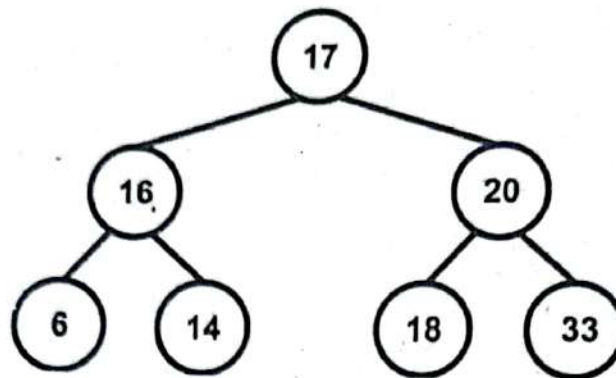
10.3.2.3 Insertion Operation on BST

In a binary search tree, a new node is always inserted at the leaf. We traverse the tree from root to leaf. We search the location in the tree (i.e. the leaf node) where the new node is to be inserted. If the value of the new node already exists in any node, then the insertion operation is terminated; otherwise, the new node is inserted at the point the search terminates.

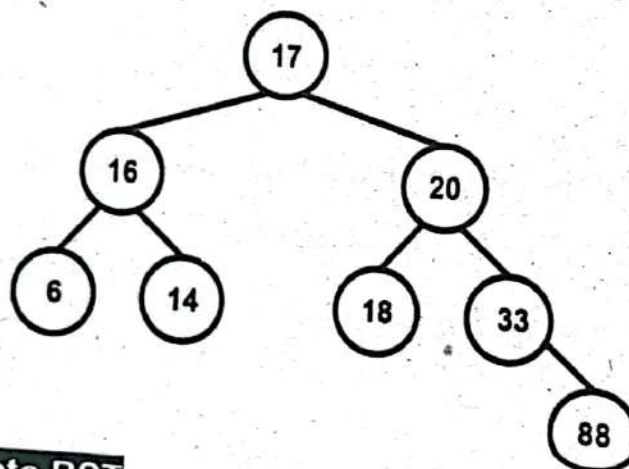
The following steps are taken to insert a new node into the binary search tree:

- 1- Create a new node with a given value and set its left and right child nodes to NULL.
- 2- Check whether the tree is empty.
- 3- If the tree is empty, then set root to a new node.
- 4- If the tree is not empty, then search the value of the new node in the tree. If a value already exists in the tree, then terminate the insertion operation; otherwise, insert the value at the point the search terminates.

Consider the following binary search tree:



Suppose we need to insert a new node with value 88 into this binary search tree. To insert this node, firstly this node is searched (with the reference to its data value) in the tree. If the node is not found, then it is inserted at the point the search process terminates. The above BST, after insertion of a new node with data value 88, is shown in the following figure:



Algorithm – Insertion Into BST

Write an algorithm that inserts a new node in a binary search tree.

1. START
2. INPUT value for the new node to insert, in variable N
3. IF the BST is empty THEN
 - Create a new node and set this node as root
 - Return i.e. stop the insertion process
 ELSE
 - Search value N in the Tree
 - IF the node with value N already exists in the tree THEN
 - PRINT "Node already exists"
 - Return
 ELSE
 Insert the node at the point the search terminates
 END IF
 END IF
 - 4. END IF
EXIT

Complexity Analysis of Insertion Operation on BST

The insertion operation of a binary search tree is performed along with the search operation. The new node is inserted at the point where the search operation terminates. Therefore, the average-case time of insertion operation is $\Theta(\log n)$, while the worst-case time complexity is $O(n)$.

The space complexity of the insertion operation is $O(n)$.

Program Code 10.8

// Program to create and insert a new node in the binary search tree

```
#include <iostream.h>
#include <conio.h>

struct node
{
    int data;
    node *right;
    node *left;
};

class bst
{
private:
    node *start, *cur, *temp, *parent;
public:
    bst(void)
    {
        start = NULL;
    }
    void create(int);
    void insert(int);
    void inorder(void);
    void inorder(node*);
};

void main(void)
{
    bst obj;
    int x[] = {22, 33, 21, 76, 88, 43, 34, 11, 89, 55};
    clrscr();
    for(int i = 0; i <= 9; i++)
        obj.create(x[i]);
    cout << "Data of tree in order \n";
    obj.inorder();
    obj.insert(99);
    cout << "\nData of tree in order after inserting node of value 99:\n";
```



```

obj.inorder();
obj.insert(2);
cout<<"\nData of tree in order after inserting node of value 2:\n";
obj.inorder();
getch();
} //end of main()

```

```

// member function to create tree
void bst::create(int n)
{

```

```

    // to create root node and assign data to it
    if(start == NULL)
    {

```

```

        start = new node;
        start->data = n;
        start->left = start->right = NULL;
    }

```

```

    else
    {

```

```

        cur = start;

```

```

        // go to proper position to add node
        while(cur!=NULL)
        {

```

```

            if(cur->data == n)
            {

```

```

                cout<<"Data already exists"<<endl;
                return;
            }

```

```

            if(cur->data < n)
            {

```

```

                parent = cur;
                cur = cur->right;
            }

```

```

            else
            {

```

```

                parent = cur;
                cur = cur->left;
            }
        }

```

```

        // create new node and assign data to it
        temp = new node;

```

```

        temp->data = n;
        temp->left = temp->right = NULL;
        if(parent==NULL)

```

```

            parent = temp;
        else if(parent->data>n)

```

```

    }
    //end of create()

    //member function to call the recursive function "inord"
    void bst::inorder()
    {
        inorder(start);
    } //end of inorder()

    // definition of inorder recursive function
    void bst::inord(node *ss)
    {
        if(ss!=NULL)
        {
            inorder(ss->left);
            cout<<ss->data<<" ";
            inorder(ss->right);
        }
    } //end of inord()

```

```

    // definition of insert() function.
    void bst::insert (int n)
    {
        if(start== NULL)
        {
            start = new node;
            start->data = n;
            start->left = start->right = NULL;
        }
        else
        {
            cur = start;

            // search value in the tree
            while(cur!= NULL)
            {
                if(cur->data == n)
                    return;
                if(cur->data < n)
                {
                    parent = cur;
                    cur = cur->right;
                }
                else
                {

```



```

        parent = cur;
        cur = cur->left;
    }
} // end of while loop

//Insert the value at the point the search terminates
temp = new node;
temp->data = n;
temp->left = temp->right = NULL;
if(parent == NULL)
    parent = temp;
else if(parent->data > n)
    parent->left = temp;
else
    parent->right = temp;
}
} //end of insert ()

```

Output of the program

Data of tree in order

11 21 22 33 34 43 55 76 88 89

Data of tree in order after inserting node of value 99:

11 21 22 33 34 43 55 76 88 89 99

Data of tree in order after inserting node of value 2:

2 11 21 22 33 34 43 55 76 88 89 99

In the above program, the while loop is used in the insert() function to search the node with value 'n' in the tree. If this loop ends in a normal way, then it means that the node with value 'n' is not found in the tree. The value of pointer 'cur' will be NULL, while the value of the pointer 'parent' will be the address of the leaf node where the search process is terminated. After termination of the loop, a new node is created and attached to the tree.

10.3.2.4 Deletion Operation on BST

The deletion operation on binary search trees is more complicated than search and insertion operations. Each node of BST has a parent except the root node. Therefore, deleting any internal node could affect all subtrees of that node. The value of the node that is to be deleted is searched in the binary search tree and if found, it is deleted.

To delete a node from BST, there are three possible cases to consider:

Case 1: Deleting a leaf node (A node with no children)

Case 2: Deleting a node with one child

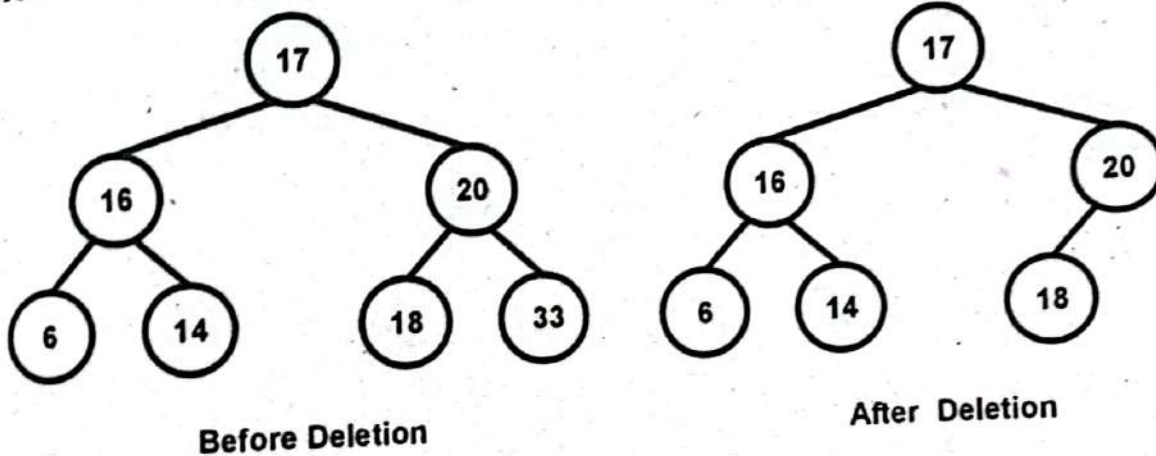
Case 3: Deleting a node with two children

Deleting a Leaf Node

Deleting a leaf node is a simple case because this node has no children. Following steps are performed to delete a leaf node:

- Find the leaf node to be deleted using a search operation.
- If the leaf node to be deleted is the left child of its parent node, then assign a NULL value to the left pointer of its parent node; otherwise, assign a NULL value to the right pointer of its parent node.

Suppose we want to delete a leaf node of value 33 from the BST as shown in the following left side figure. Move to this node using a search operation. It is the right child of its parent node (i.e. node 20). Assign a NULL value to the right pointer of its parent node. The leaf node will be disconnected from the tree and thus it is automatically deleted. The binary search trees before and after deletion operation are shown in the following figures:



Algorithm – Deleting a Leaf Node from BST

Write an algorithm for deleting a leaf node from a binary search tree.

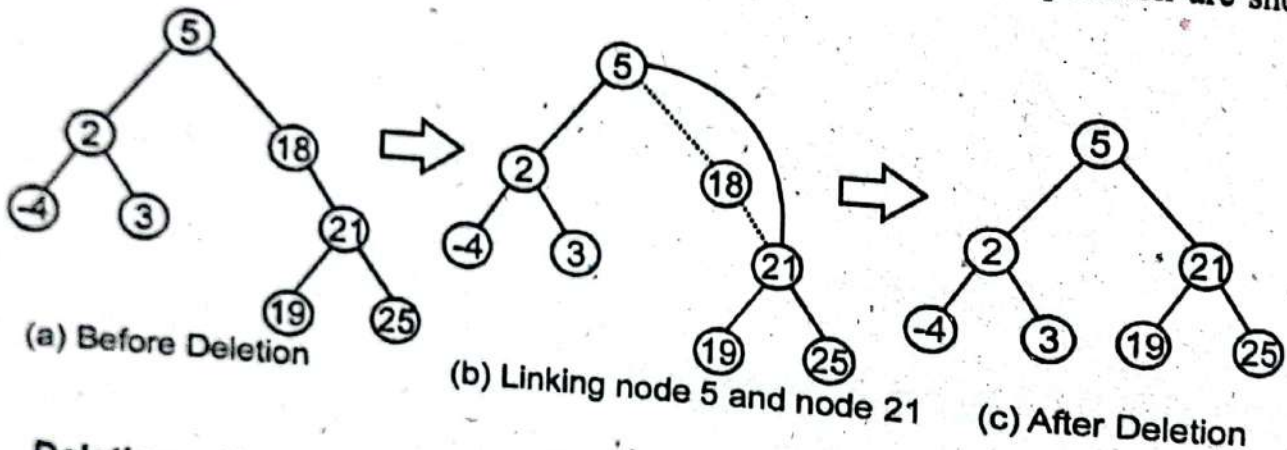
- 1- START
- 2- INPUT value of the node to be deleted, in variable N
- 3- Find the node to be deleted that has value N in the Tree
- 4- IF node is not found in the tree THEN
 PRINT "Node not found"
 Return
 END IF.
- 5- IF node is found in the tree THEN
 IF node is the left child of the tree THEN
 Assign a NULL value to the left pointer of its parent node
 ELSE IF node is the right child of the tree THEN
 Assign a NULL value to the right pointer of its parent node
 END IF
 END IF
- 6- END IF
 EXIT

II) Deleting a Node with One Child

Deleting a node with one child is also a simple case. This case is similar to delete a node from a linked list. Following steps are performed to delete a node with one child:

- Find the node to be deleted using the search operation.
- If the node to be deleted is found and has only one child which can be a left child or right child.
- If the node to be deleted has a left child, then we link the left child directly to the parent of the deleted node.
- If the node to be deleted has a right child, then we link the right child directly to the parent of the deleted node.

Suppose we want to delete a node 18 from the BST as shown in the following figure (a). This node has only one child (i.e. right child) which is node 21. Node 18 is the right child of its parent node i.e. node 5. We simply link node 5 (right child node of node 18) to the right pointer of node 18 (parent node of node 18). The node 18 will be disconnected from the tree and thus it is automatically deleted. The binary search trees before and after deletion operation are shown in the following figures:

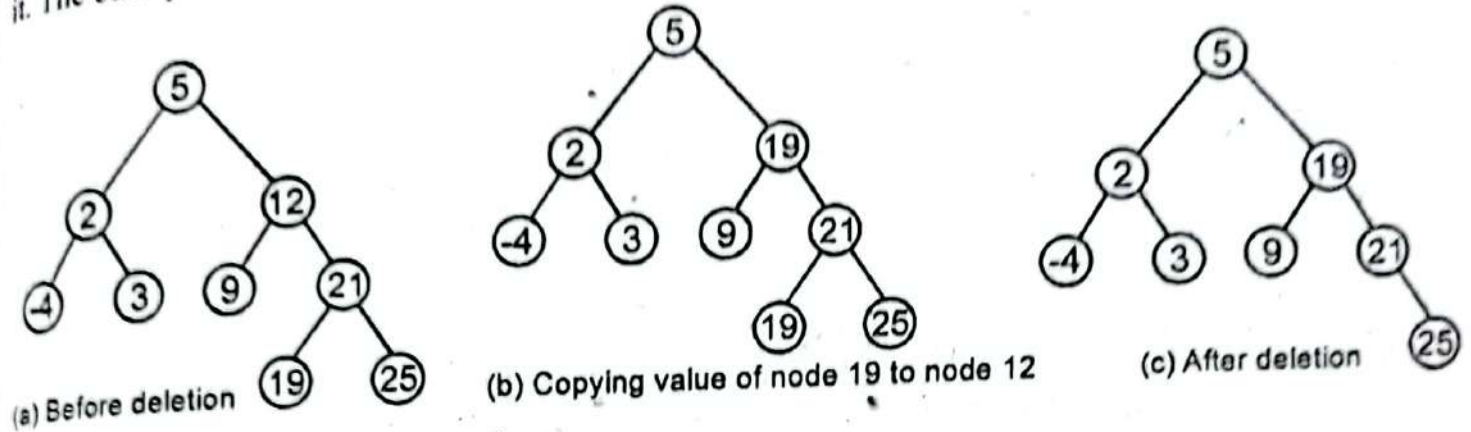


III) Deleting a Node with Two Children

This is a difficult case as we need to deal with two subtrees. Following steps are performed to delete a node with two children:

- 1- Find the node to be deleted using the search operation.
- 2- If it has two children, then find the node in the left subtree that has the largest value.
OR find the node in the right subtree that has the smallest value.
- 3- Copy the value of the node found in step-2 to the deleting node. Suppose we find the node in the right subtree that has the smallest value.
- 4- Now delete the node whose value is copied to the deleting node in step-3 (the node that has the smallest value in the right subtree of deleting node).
 - If it is a leaf node, then use deletion case 1.
 - If it has one child, then use deletion case 2.
- 5- Repeat the same process until the node is deleted from the tree.

Suppose we want to delete a node 12 from the BST as shown in the following figure (a). This node has two children. Node 12 is the right child of its parent node i.e. node 5. First of all, we will find the node that has the largest value in the left subtree of node 12 or the node that has the smallest value in the right subtree of node 12. Let us find the node that has the smallest value in the right subtree of node 12. It is node 19. We copy the value of this node to node 12 and delete it. The binary search trees before and after deletion operation are shown in the following figures:



If we search the smallest value in the right subtree and the root node of this subtree has a left child, then the smallest value will be in the left branch of the right subtree. We have to search all next coming left child nodes of parent nodes successively. However, if the root node of the right subtree has no left child, then the value of the root of this subtree will be the smallest value.

If we search the largest value in the left subtree and the root node of this subtree has a right child, then the largest value will be in the right branch of the left subtree. We have to search all the next coming right child nodes of parent nodes successively. However, if the root node of the left subtree has no right child, then the value of the root of this subtree will be the largest value.

Complexity Analysis of Deletion Operation on BST

The deletion operation of a binary search tree is performed along with the search operation. The node is deleted from the BST if found. Like insertion operation, the average-case time of deletion operation is $\Theta(\log n)$, while the worst-case time complexity is $O(n)$.

The space complexity of the deletion operation is $O(n)$.

Program Code 10.9

// Program to create and delete a node from the binary search tree

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
struct node
```

```
{
```

```
    int data;
```

```
    node *right;
```



```

        node *left;
    };

    class bst
    {
    private:
        node *start, *cur, *temp, *parent;
    public:
        bst(void)
        {
            start = NULL;
        }

        void create(int);
        void delete_node (void);
        void delete_leaf_node (node*, int);
        void delete_onechild_node (node*, int);
        void delete_twochildren_node (node*, int);
        void preorder (void);
        void preord(node*);
    };

    bst obj; // object of class 'bst' is declared as global

```

```

void main(void)
{
    int x[] = {22, 33, 21, 76, 88, 43, 34, 11, 89, 55};
    clrscr();
    for(int i = 0; i<=9; i++)
        obj.create(x[i]);
    cout<<"\nData of tree in preorder "<<endl;
    obj.preorder();
    obj.delete_node();
} //end of main()

```

// member function to create tree

```
void bst::create(int n)
```

```

{
    int col = 40, row = 1;

    // to create root node and assign data to it
    if(start == NULL)
    {
        start = new node;
        start->data = n;
        start->left = start->right = NULL;
        gotoxy(col, row);
        cout<<n;
    }
}

```

else

{

cur= start;

// go to proper position to add node

while(cur!=NULL)

{

if(cur->data == n)

{

cout<<"Data already exists"<<endl;

return;

}

if(cur->data < n)

{

parent = cur;

cur = cur->right;

row+= 2;

col+= 6;

}

else

{

parent = cur;

cur = cur->left;

row+= 2;

col-= 6;

}

}

// create new node and assign data to it

temp = new node;

temp->data = n;

temp->left = temp->right = NULL;

gotoxy(col, row);

cout<<n;

if(parent==NULL)

parent = temp;

else if(parent->data>n)

parent->left = temp;

else

parent->right = temp;

}

//end of create()

//member function to call the recursive function preorder

void bst::preorder()

{

preord(start);

//end of preorder()


```

//preord recursive function to visit the tree in a preorder traversal
void bst::preord(node *ss)
{
    if(ss!=NULL)
    {
        cout<<ss->data<<" ";
        preord(ss->left);
        preord(ss->right);
    }
} //end of preord()

```

```

// function definition for deleting node from BST.
void bst::delete_node()
{
    int val;
    cout<<"\nEnter value of leaf node to delete:";
    cin>>val;
    obj.delete_leaf_node (start, val);
    cout<<"\nEnter value of one child node to delete:";
    cin>>val;
    obj.delete_onechild_node (start, val);
    cout<<"\nEnter value of two child nodes to delete:";
    cin>>val;
    obj.delete_twochildren_node (start, val);
} //end of delete_node ()

```

```

// function definition to delete leaf node from BST.
void bst::delete_leaf_node (node *ds, int n)
{
    if(ds == NULL)
    {
        cout<<"Tree is empty";
        return;
    }
    else // find the leaf node.
    {
        cur = parent= ds;
        // search value
        while(cur!=NULL)
        {
            if(cur->data == n)
            {
                if (cur->left ==NULL && cur->right ==NULL)
                {
                    break;
                }
                if(cur->data < n)
                {
                    // if leaf node found then loop break
                }
            }
        }
    }
}

```

```

        {
            parent = cur;
            cur = cur->right;
        }
        else
        {
            parent = cur;
            cur = cur->left;
        }
    } //end of while loop
    if(cur == NULL)
    {
        cout<<"Leaf node not found";
        getch();
        return;
    }
} //end of else part ()

// Delete leaf node
if(n > parent->data)
    parent->right = NULL;
else
    parent->left = NULL;
cout<<"\nData of tree in preorder after deleting the node " <<endl;
obj.preorder();
getch();
} //end of delete_leaf()

// function definition to delete one child node from BST.
void bst::delete_onechild_node (node *ds, int n)
{
    if(ds == NULL)
    {
        cout<<"Tree is empty";
        return;
    }
    else // find the one child node
    {
        cur = parent = ds;
        // search value
        while(cur != NULL)
        {
            if(cur->data == n)
            {
                // checks one child node that has left child but no right child
                if (cur->left != NULL && cur->right == NULL)
                    break;
                // if one child node found then loop break
            }
        }
    }
}

```



```

// checks one child node that has right child but no left child
else if (cur->left == NULL && cur->right != NULL)
    break; // If one child node found then loop break

```

```

    if (cur->data < n)
    {
        parent = cur;
        cur = cur->right;
    }
    else
    {
        parent = cur;
        cur = cur->left;
    }
} //end of while loop
if (cur == NULL)
{
    cout << "One child node not found";
    getch();
    return;
} //end of else part ()

//Delete one child node
if (cur->left != NULL)
    parent->left = cur->left;
else
    parent->right = cur->right;
cout << "\nData of tree in preorder after deleting the node " << endl;
obj.preorder();
getch();
} //end of delete_onechild_node()

```

```

// function definition to delete two children node from BST.
void bst::delete_twochildren_node (node *ds, int n)
{

```

```

    int mln;
    if (ds == NULL)
    {
        cout << "Tree is empty";
        return;
    }

```

```

    else
    {

```

```

        cur = parent = ds;

```

```

        // search value

```

```

        while (cur != NULL)

```

```

if(cur->data == n)
    // check two children node that has left child and right child
    if (cur->left!=NULL && cur->right!=NULL)
        break; // if node of two child nodes found then loop break
    if(cur->data < n)
    {
        parent = cur;
        cur = cur->right;
    }
    else
    {
        parent = cur;
        cur = cur->left;
    }
} //end of while loop

if(cur == NULL)
{
    cout<<"Node with two children not found";
    getch();
    return;
}
else // find the smallest value in right subtree
{
    /* cur contains address of the node to be deleted
    value or address of cur is assigned to temp pointer */
    temp = cur;
    // now address of right child of node to be deleted is assigned to cur
    cur = cur->right;
    min = cur->data;
    // find smallest value in right subtree
    while(cur!=NULL)
    {
        parent = cur;
        if(cur->data < min)
            min = cur->data;
        cur = cur->left;
    } //end of while loop
}

// Copy the value smallest node in right subtree into searched node with two child nodes
if(parent->left==NULL)
    temp->data = min; // copy the smallest into the node to be deleted

// delete the node that has smallest value in right subtree

```



```

        if(parent->left==NULL && parent->right==NULL) // if node is leaf node
            obj.delete_leaf_node(temp->right, min);
        else // if node is one child node
            obj.delete_onechild_node(temp->right, min);
    }
} //end of delete_twochildren_node()

```

Output of the program

```

      22
     21  33
    11   76
      43  88
     34  55  89

```

Data of tree in preorder

22 21 11 33 76 43 34 55 88 89

Enter value of leaf node to delete:11

Data of tree in preorder after deleting the node

22 21 33 76 43 34 55 88 89

Enter value of one child node to delete:33

Data of tree in preorder after deleting the node

22 21 76 43 34 55 88 89

Enter value of two child nodes to delete:22

Data of tree in preorder after deleting the node

34 21 76 43 55 88 89

10.4 EXPRESSION BINARY TREE

A binary expression tree is a specific kind of a binary tree. It is used to represent expressions. It uses arithmetic operators and operands as node values. The internal nodes represent the operators of the expression. The leaves of an expression tree represent the operands of the expression which may be either constants or variables. Most operators represent binary operations and thus most internal nodes have two children. However, this is not always the case. Unary operators, such as unary minus, will have only a single child node representing the single operation of such operator.

Expression trees are useful for representing and evaluating expressions. They are used to convert infix expressions into prefix expressions.

10.4.1 Constructing an Expression Binary Tree

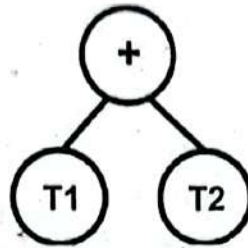
Consider the following arithmetic expression:

$$(a + (b * c)) + (((d * e) + f) * g)$$

To construct the expression binary tree of the arithmetic expression, the expressions within parenthesis are divided into subtrees. The left part will be the left subtree and the right part will be the right subtree of the expression binary tree. Thus, the given expression is rewritten as:

$(a+(b*c))$	as left subtree say T1
$((d*e)+f)*g$	as right subtree say T2
$+$	root of T1 and T2

The binary tree is constructed as:



The subtrees T1 and T2 are further divided as follows:

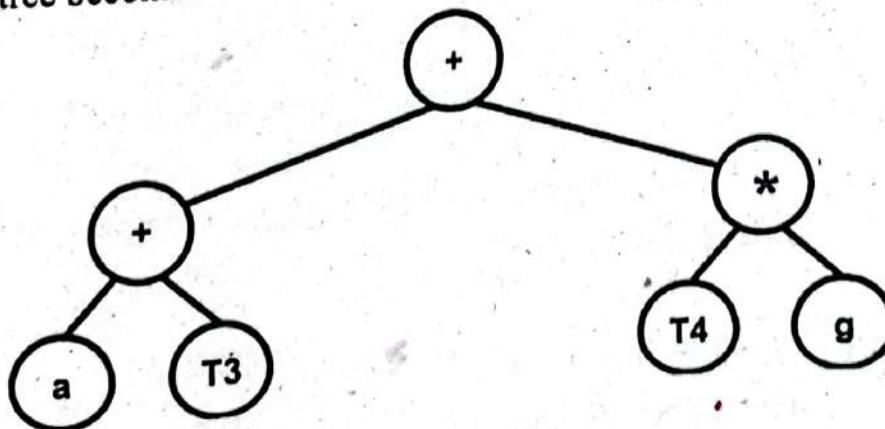
$(a+(b*c))$ is divided as:

- "a" as the left subtree
- "b*c" as a right subtree T3
- "+" as the root of these subtrees

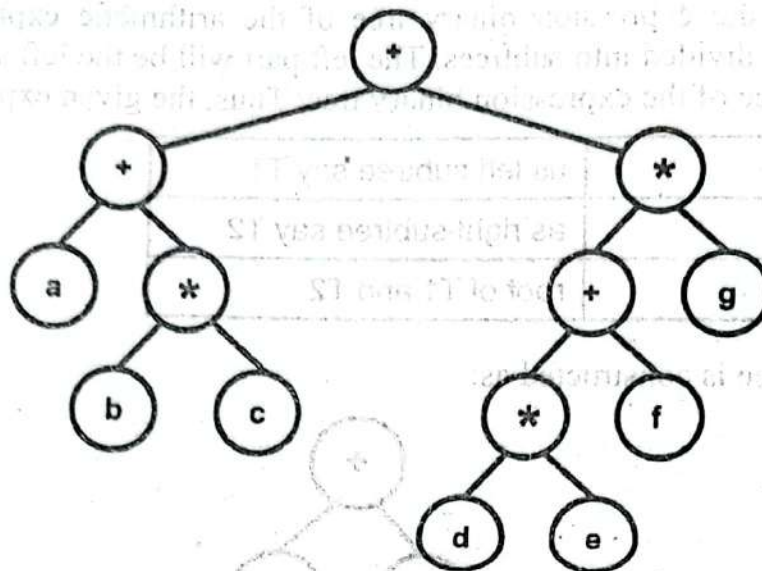
$((d*e)+f)*g$ is divided as:

- " $((d*e)+f)$ " as left subtree T4
- "g" as a right child, and
- "*" as the root of these subtrees.

The above tree becomes as:



The T3 and T4 are further expanded and the final expression binary tree is as given below:



To convert the expression to prefix Polish notation, the preorder traversal is performed. By using the preorder traversal of the above tree, the expression is obtained in prefix notation as:

$++a*b*c*+*d*efg$

To convert the expression to postfix notation, the postorder traversal is performed. By using the postorder traversal of the above tree, the expression is written in postfix Polish notation as:

$abc*+de*f+g*+$

10.4.2 Constructing Binary Expression Tree using Stack

A binary expression tree can also be constructed by using a stack. To construct a binary expression tree, follow these steps:

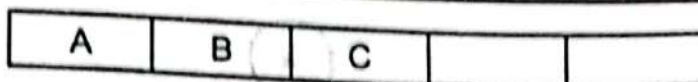
1. Convert the infix expression into postfix form.
2. Read the postfix expression from left to right.
3. If an operand is encountered, push it onto the stack.
4. If an operator is read, pop the first two operands from the stack and create a tree with the read operator as its root node.
5. Store the addresses of the newly created tree onto the stack.
6. Repeat steps 3 and 5 until the end of the expression.

Consider the conversion of the expression $[A+(B-C)]*[(D-E)]$

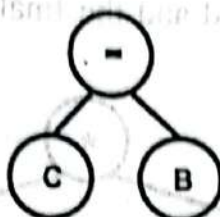
★ The expression is written in postfix expression as:

$A B C - + D E - *$

★ Read the expression from left to right. Push the first three operands A, B, and C into the stack as they are read (scanned). The stack will be as shown below:



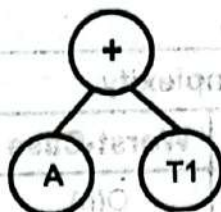
- ★ Next, the operator "-" is read. Pop the first two operands C and B from the stack. The first tree T1 is created with the operator at its root node and the two operands as its leaves.



- ★ The address of the tree is then pushed onto the stack. The stack will be as shown below:



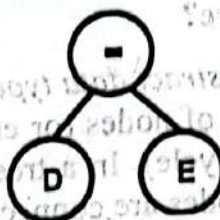
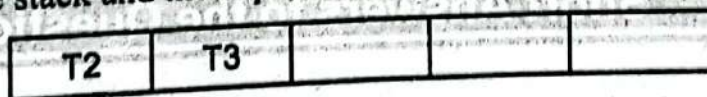
- ★ When the operator "+" is read, T1 and A are popped from the stack and the tree T2 is created say with "+" operator as root, T1, and A as its leaves. The address of T2 is pushed onto the stack. The stack and the expression tree will look like this:



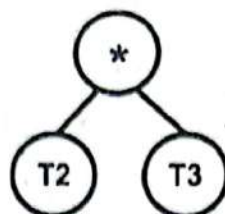
- ★ Next two operands D and E are pushed onto the stack as shown below:



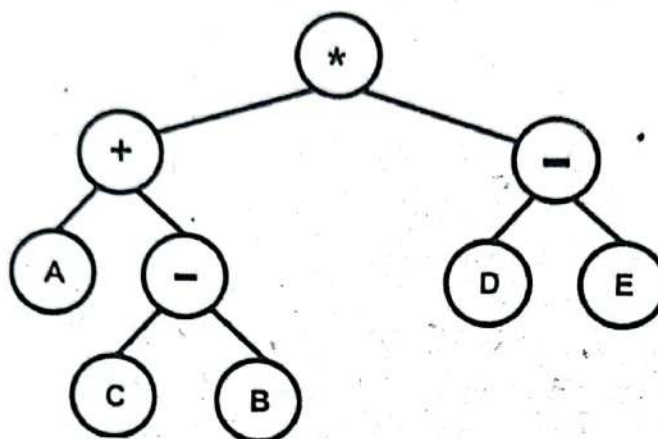
- ★ When the operator "-" is read, both D and E are popped. Tree T3 is created with "-" operator as root and D and E as its leaves. The address of the tree T3 is pushed onto the stack. The stack and the expression tree T3 will be as shown below:



- ★ When the next operator "*" is read, both T2 and T3 are popped and tree T4 is created with "*" operator as its root and its address is pushed onto the stack.



- ★ All subtrees are combined and the final tree is created as shown in the following figure:



Summary --- Complexities of Binary Search Tree

Binary Search Tree (BST)

Operation	Time Complexity		Space Complexity
	Average-Case	Worst-Case	Worst-Case
Search	$\Theta(\log n)$	$O(n)$	$O(n)$
Traversal	$\Theta(\log n)$	$O(n)$	$O(n)$
Insertion	$\Theta(\log n)$	$O(n)$	$O(n)$
Deletion	$\Theta(\log n)$	$O(n)$	$O(n)$

Short Answers to the Questions



What is a tree in data structure?

Tree is the most widely used *abstract data type* (ADT) or data structure. It is a non-linear data structure. It is a collection of nodes (or entities) that are organized in a hierarchical structure (without having any cycle). In a tree data structure, every individual entity or element is called a *node*. The nodes are connected by edges (The connection line between two nodes is called an *edge*). Each node contains a value or data. The first node of the tree is called the *root node* and it may contain several links.

Define the root node.

The topmost node in the tree that has no parent node is called the *root node*. It represents the beginning of the tree.

Define parent node.

A node that has one or more child nodes is called a *parent node*. It is also known as an *ancestor node*. It is always above its child nodes.

What is child node?

The node that is directly connected to a parent node is called the *child node*.

What is the difference between the depth of a node and the degree of a node?

In a tree data structure, the total number of edges (i.e. length of path) from the root node to a particular node is called the *depth* of that node. The total number of children (child nodes) of a node is called the *degree of that node*.

What do you know about the binary tree?

A tree in which each node can have a maximum of two children is called a *binary tree*. In a binary tree, one child lies on the left side and the other lies on the right side of the tree or subtree. A binary tree may be empty or non-empty but a general tree cannot be empty.

What is the difference between a full tree and a full binary tree?

A tree whose all-internal nodes have the same degree and all its terminals are at the same level is called a *full tree*. A binary tree in which every node other than leaves has two children is called a *full binary tree*. In this type of binary tree, every node has either 0 or 2 children.

What is the difference between internal node and external node in a binary tree?

The node that has two children is called an *internal node* and the node that has no child is called an *external node*.

What is the expression binary tree?

The binary tree which uses arithmetic operators and operands as node values is called the *expression binary tree*. The internal nodes represent the operators of the expression. The leaves of an expression tree represent the operands of the expression which may be either constants or variables.

What is the difference between a binary tree and a tree?

A binary tree may be empty, while a tree cannot be empty. No node in a binary tree may have a degree more than 2, whereas there is no limit on the degree of a node in a tree. The subtrees of a binary tree are ordered, whereas subtrees of a tree are not ordered.

There are 8, 15, 13, and 14 nodes in 4 different trees. Which one of them can form a full binary tree?

The answer is the tree with 15 nodes. In general, there are $2^n - 1$ nodes in a full binary tree. Full binary trees always contain an odd number of nodes, so there cannot be full binary trees with 8 or 14 nodes. Moreover, with 13 nodes we can form a complete binary tree but not a full binary tree. Thus, the correct answer is 15.

Why it is said that searching a node in a binary search tree is more efficient than that of a simple binary tree?

In a binary search tree, the nodes are arranged in such a way that the left node has a smaller data value than the root node value and the right nodes have a larger value than that of the root. Because of this, while searching any node the value of the target node will be compared with the parent node, and accordingly either left sub-branch or right sub-branch will be searched. So, one has to compare only particular branches. Thus searching becomes efficient.

EXERCISE

Q# 1 Select the correct answer from the given multiple choices.

- Which of the following is true about a tree?
 - It is non-linear data structure
 - It is a set of vertices
 - It is a set of edges
 - All
- The topmost node in the tree that has no parent node is called:
 - Head node
 - Root node
 - Terminal node
 - Parent node
- A node that has one child or more children is called:
 - Ancestor node
 - Parent node
 - Superior node
 - All
- The node having no successor or child is called:
 - Child node
 - Root node
 - Terminal node
 - Parent node
- The height of an empty tree is:
 - 1
 - 0
 - NULL
 - None
- The number of children of a node is called:
 - Depth of node
 - Height of node
 - Degree of node
 - None
- The tree whose root is its only node is called:
 - Singleton tree
 - Binary tree
 - Incomplete tree
 - Empty tree
- A tree in which each node can have a maximum of two children is called:
 - General tree
 - Binary tree
 - Bi-Tree
 - None
- A tree in which each node must have two children and all terminals must be at same level is called:
 - General tree
 - Binary tree
 - Full binary tree
 - None
- When inorder traversal of a tree resulted E A C K F H D B G; the preorder traversal would return:
 - FAEKCDHBG
 - FAEKCDHGB
 - EAFKHDCBG
 - FEAKDCHBG

To represent hierarchical relationship between elements, which data structure is suitable?

(a) Array (b) Priority Queue (c) Tree (d) All

- (a) Deque (b) Priority (c) Tree (d) All
- every tree whose every node has either zero or two children is called:

A binary tree whose every node has either zero or two children is called:

- (a) Complete binary tree (b) Binary search tree
(c) Extended binary tree (d) None

The postorder traversal of a binary tree is DEBFCA. Find out the preorder traversal:

- (a) ABFCDE (b) ADBFEC (c) ABDECF (d) ABDCEF

(a) The inorder traversal of tree will yield a sorted listing of elements of tree in: (b) Binary search tree (c) B+ tree (d)

- (a) Binary trees (b) Binary search trees (c) Heaps (d) None

(a) Binary
A terminal node in a binary tree is called:

- (a) Root (b) Leaf (c) Child (d) Branch

In an extended-binary tree, nodes with 2 children are called:

- In an extended-binary tree, nodes with 2 children are called:
- (a) Interior nodes (b) Domestic nodes (c) Internal nodes (d) Inner nodes

Which one of the following indicates pre-order traversal?

- Which one of the following indicates pre-order traversal?
- (a) Left sub-tree, Right sub-tree and root (b) Right sub-tree, Left sub-tree and root
(c) Root, Left sub-tree, Right sub-tree (d) Right sub-tree, root, Left sub-tree
- Which of the following statement about binary tree is **CORRECT**?

(c) Root, Left sub-tree, Right sub-tree

Which one of the following statement about binary tree is CORRECT?

- (a) Every binary tree is either complete or full
(b) Every complete binary tree is also a full binary tree
(c) Every full binary tree is also a complete binary tree
(d) A binary tree cannot be both complete and full

(d) A binary tree cannot be both complete and full.
The number of edges from the root to the node is called _____ of the tree.

(b) Depth (c) Length (d) None

- (a) Height (b) Depth (c) Length (d) None

A tree is called full binary tree if:

- (a) Each node has exactly zero or two children
(b) Each node has exactly two children
(c) All the leaves are at the same level
(d) Each node has exactly one or two children

(d) Each node has exactly one or two children
The worst-case time complexity of preorder traversal algorithm in BST is: (c) $\Theta(n \log n)$ (d)

- The worst-case time complexity of preorder traversal is: (a) $O(n)$ (b) $\Theta(\log n)$ (c) $\Theta(n \log n)$ (d) $O(n^2)$

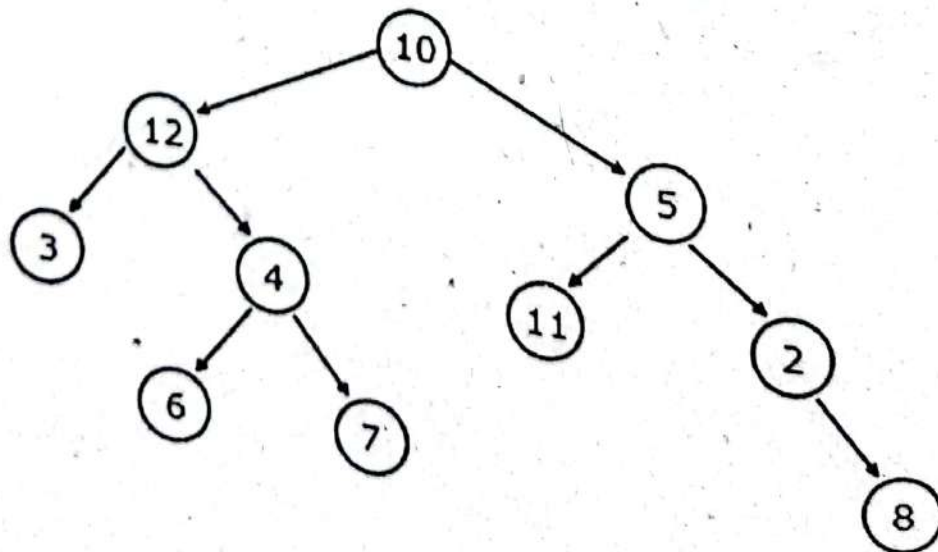
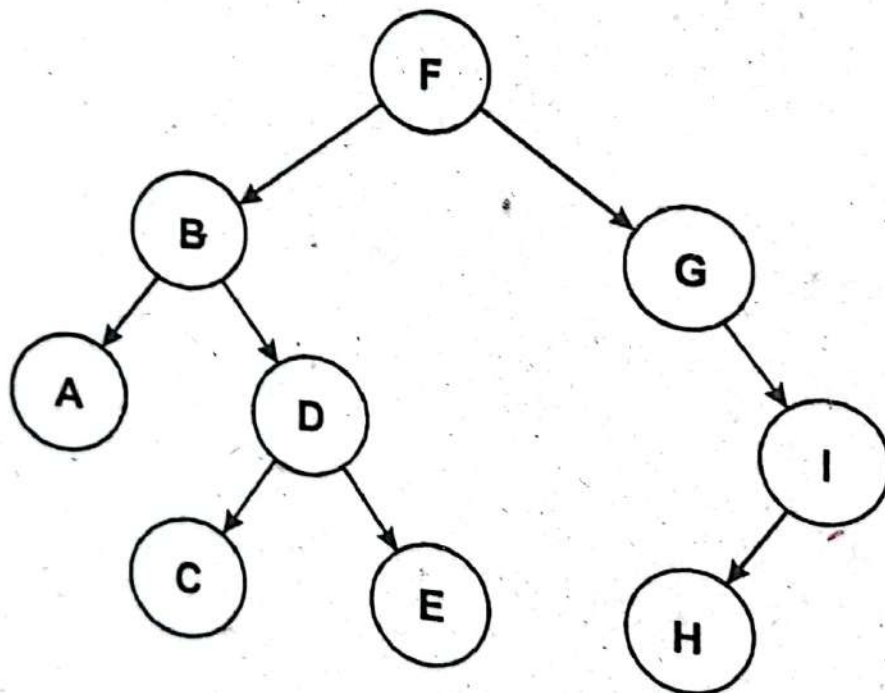
(a) $O(n)$ (b) $\Theta(\log n)$ (c) $O(1)$
The worst-case time complexity of insertion algorithm in BST is:

- The worst-case time complexity of insertion algorithm is
- (a) $O(n)$ (b) $O(\log n)$ (c) $O(1)$ (d) $O(n^2)$

Answers of MCQs

01 (a)	02 (b)	03 (d)	04 (c)	05 (a)	06 (c)	07 (a)	08 (b)	09 (c)	10 (c)
11 (c)	12 (c)	13 (c)	14 (b)	15 (b)	16 (c)	17 (c)	18 (c)	19 (b)	20 (a)
21 (a)	22 (a)								

- Q.2 Write a C++ program to find the depth or height of a tree.
- Q.3 Write a C++ program to find the smallest value in a binary search tree.
- Q.4 Draw a diagram of a binary tree, list the nodes in:
(a) pre-order (b) in-order (c) post-order
- Q.5 Give rules for the standard method of converting a general tree to a binary tree. Illustrate with an example.
- Q.6 Draw a complete binary tree with exactly six nodes. Put a different value in each node. Then draw an array with six components and show where each of the six nodes values would be placed in the array.
- Q.7 What are the results of inorder, preorder, and postorder traversal of the following trees:



Binary search tree is an excellent data structure to implement associative arrays, maps, sets, and similar interfaces. It is efficient only when it is balanced. The insertion and deletion operations can lead to a highly unbalanced tree. For example, if we insert 'n' elements with values that are in strictly increasing or decreasing order, the complexity will be $O(n^2)$. On the other hand, if we can keep the height to $O(\log n)$, as it is for a perfectly balanced tree, then the complexity is bounded by $O(n \log n)$.

The solution is to dynamically rebalance the binary search tree during insertion or deletion operations. Because of the importance of binary search trees, researchers have developed many different algorithms for keeping trees in balance, such as AVL trees, red-black trees, splay trees, and B-trees.

11.1 AVL Tree

AVL tree is a special type of binary search tree that is "almost" balanced. In an AVL tree, the heights of the left and right subtrees of any node differ by at most one. Due to this property, the AVL tree is also called the *height balanced tree*. If at any time they differ by more than one, rebalancing is done to restore this property. AVL tree was introduced by two mathematicians Adelson-Velskii and Landis (AVL) in 1962, so it is named AVL in honor of its inventors.

In an AVL tree, every node is associated with additional information known as a *balance factor*. It is the difference between the heights of the left and right subtrees of the node. The node structure of the AVL tree is as follows:

Left	Node_Data	Balance_Factor	Right
------	-----------	----------------	-------

struct node

```
{
    int Node_Data;
    int Balance_Factor;
    node *Left, *Right;
};
```

The balance factor of every node of the AVL tree is either -1, 0, or +1. A tree is said to be balanced if the balance factor of each node is between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced. The rotations are performed to balance the tree.

Balance factor is calculated as:

Balance factor = height of left subtree - the height of right subtree

Consider the following binary search tree with heights of every node:

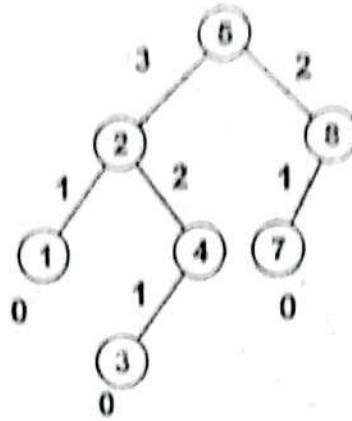


Figure 11.1: BST with heights of every node

In this tree, the balance factor for different nodes is calculated as:

- **Node with value 5 (Root node):** The height of the left subtree is 3 and the right subtree is 2. So, the balance factor of this node i.e. root node is:

$$B.F = 3 - 2 = 1$$
- **Node with value 2:** The height of the left subtree of this node is 1 and the right subtree is 2. So, the balance factor of this node is:

$$B.F = 1 - 2 = -1$$
- **Node with value 1:** The height of the left subtree of this node is 0 and the right subtree is also 0. So, the balance factor of this node is:

$$B.F = 0 - 0 = 0$$
- **Node with value 4:** The height of the left subtree of this node is 1 and the right subtree is 0. So, the balance factor of this node is:

$$B.F = 1 - 0 = 1$$
- **Node with value 8:** The height of the left subtree of this node is 1 and the right subtree is 0. So, the balance factor of this node is:

$$B.F = 1 - 0 = 1$$

Note that the balance factor of the leaf node is always 0.

11.2. The BST as shown in figure 11.1 with balance factors of every node is shown in figure

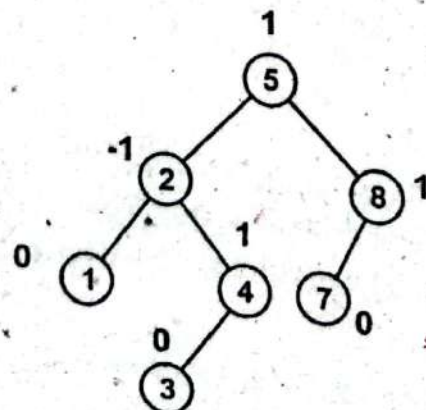


Figure 11.2: BST with balance factors of every node

The tree shown in figure 11.2 is a binary search tree and every node is satisfying the balance condition. So this tree is said to be an AVL tree. The binary tree shown in figure 11.3 is not an AVL tree because in this tree the balance factor of the root node is 2 which violates the balance factor condition.

An AVL tree is identical to a binary search tree, except that for every node in the AVL tree, the height of the left and the right subtrees can differ by at most 1. AVL tree is a self-balancing binary search tree. Every AVL tree is a binary search tree but all the binary search trees need not be AVL trees.

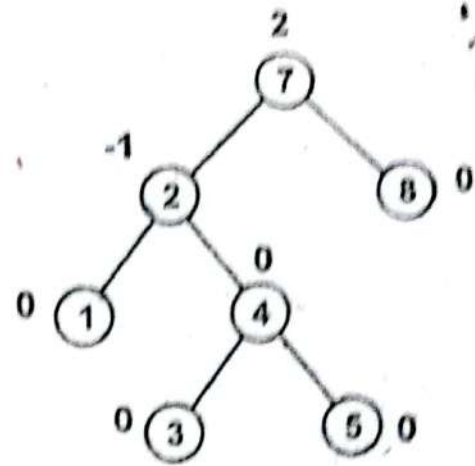
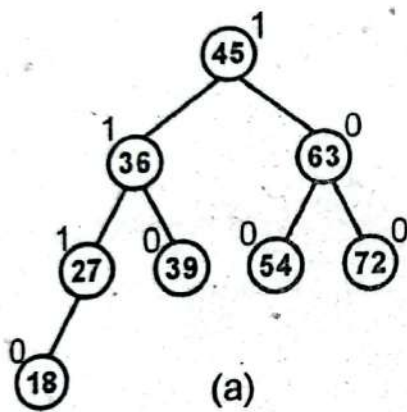


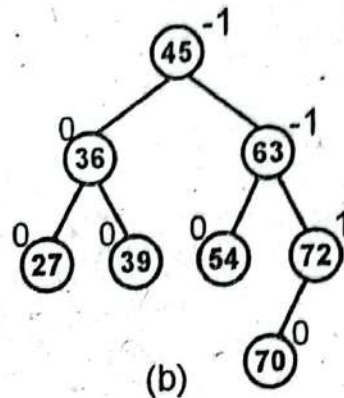
Figure 11.3: Unbalanced Binary Tree

11.1.1 Forms of AVL Trees

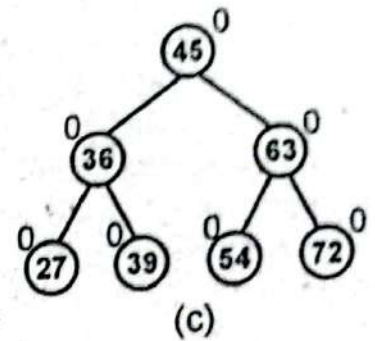
Different forms of AVL trees based on the balance factors are as follows:



(a) Left-heavy tree



(b) right-heavy tree



(c) balanced tree

Figure 11.4: Different forms of AVL trees

Left-Heavy AVL Tree

A root node with balance factor 1 is called a *left-heavy AVL tree*. It means that the left sub-tree is one level higher than the right sub-tree.

Right-Heavy AVL Tree

A root node with balance factor -1 is called a *right-heavy AVL tree*. It means that the right sub-tree is one level higher than the left sub-tree.

Balanced AVL Tree

A root node with balance factor 0 is called a *balanced AVL tree*. It means that the left sub-tree and right sub-tree have equal height.

Figure 11.4 shows the left-heavy, right-heavy, and balanced AVL trees:

11.1.2 Operations on AVL Tree

AVL tree is also a binary search tree, therefore; all the operations are performed in the same way as in a binary search tree. These operations are searching, traversing, insertion, and deletion. Searching and traversing operations do not lead to the violation property of AVL tree. However, insertion and deletion operations can violate this property, and therefore, they may require the tree to be rebalanced by one or more tree rotations.

Consider the AVL tree as shown in figure 11.5. Now insert a new node with value 14. The insertion algorithm without rebalancing will put it to the right of 13 as shown in figure 11.6. The height of this tree will become 3, and one path is longer than the others. However, it is easy to check that at each node, the height of the left and right subtrees still differ only by one. For example, at the node with value 16, the left subtree has height 2 and the right subtree has height 1, which still obeys the balance factor condition (or height invariant). This tree is also an AVL tree.

Now insert another node with value 15. It will be inserted to the right of the node with value 14. After insertion node with value 15, the tree is shown in figure 11.7.

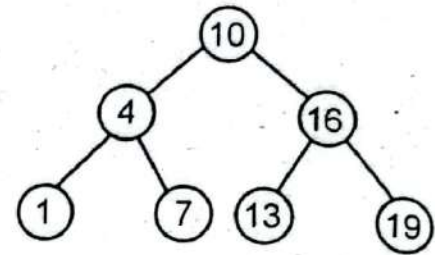


Figure 11.5: AVL Tree

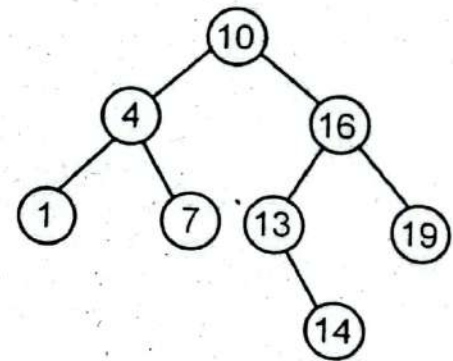


Figure 11.6: AVL Tree after insertion node 14

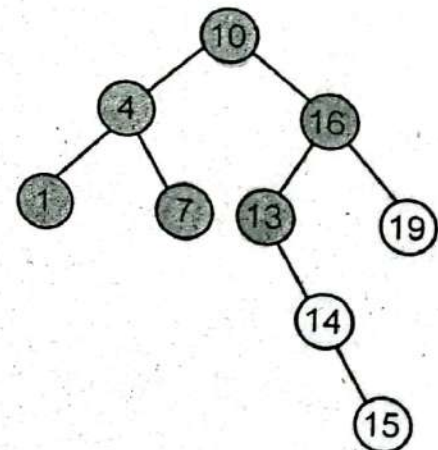
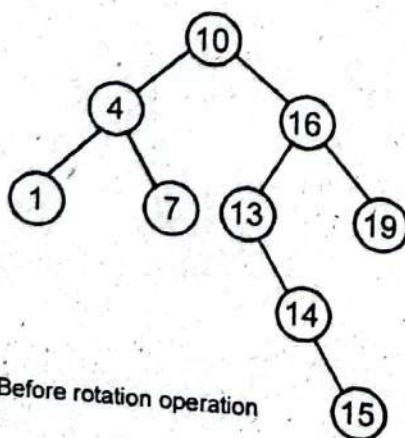
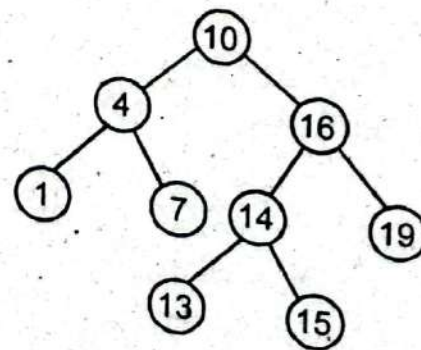


Figure 11.7: Tree after insertion node 15



Before rotation operation



After rotation operation

Figure 11.8: Unbalanced and Balanced Binary Trees

This tree is not an AVL tree because, at the node with value 13, the left subtree has height 0, while the right subtree has height 2. Similarly, at the node with value 16, the left subtree has height 3, while the right subtree has height 1. These nodes violate the balance factor condition. We, therefore, have to take steps to rebalance this tree. If we remove the node with value 14 up and node with value 13 down, the tree can be rebalanced. Actually, we will rotate the node with value 13 to left (anti-clockwise). It is the rotation operation that is performed to rebalance the tree. Figure 11.8 shows an unbalanced tree before the rotation operation and a balanced tree after the rotation operation.

11.1.3 AVL Tree Rotations

We use rotation operations to make the tree balanced. Rotation is a process of moving the nodes to either left or right to make the tree balanced. AVL tree rotations are classified into two main types:

- 1) Single Rotation
- 2) Double Rotation

11.1.3.1 Single Rotation

The *single rotation* is further divided into two types:

- i) Left Rotation (LL Rotation i.e. Left-Left Rotation)
- ii) Right Rotation (RR Rotation i.e. Right-Right Rotation)

Left Rotation (LL Rotation i.e. Left-Left Rotation)

In Left Rotation (LL Rotation), every node moves one position to left from the current position. This type of rotation is performed when we insert a new node into the right subtree and the tree becomes unbalanced. Consider the following trees:

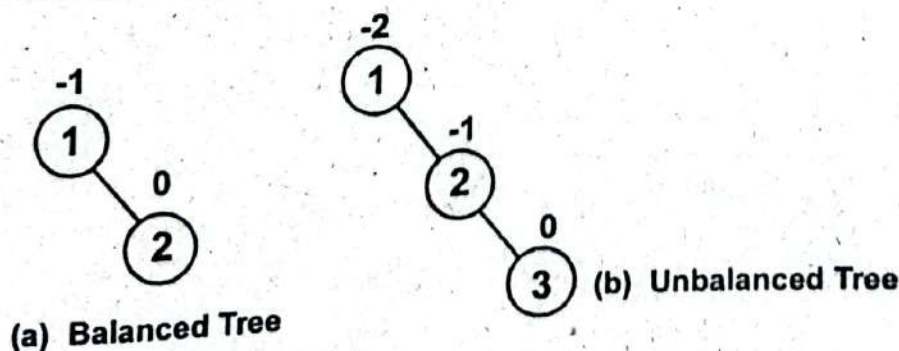


Figure 11.9: Balanced and Unbalanced Binary Trees

The tree shown in figure 11.9(a) is balanced. When a new node with value 3 is inserted into the right subtree of this tree, the tree becomes unbalanced as shown in figure 11.9(b) because the balance factor of a node (root node) with value '1' is -2. We can balance this tree by using LL

Rotation (Left Rotation) which moves nodes one position left. Figure 11.10 shows the unbalanced tree before LL Rotation and balanced tree after LL Rotation.

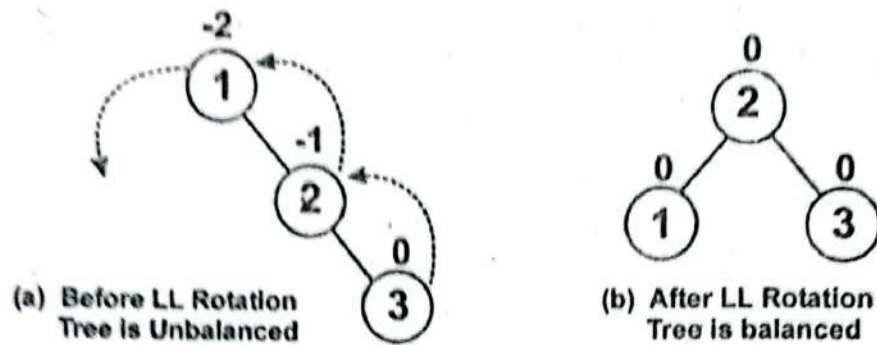


Figure 11.10: Unbalanced and Balanced Binary Trees

ii) Right Rotation (RR Rotation i.e. Right-Right Rotation)

In Right Rotation (RR Rotation), every node moves one position to the right from the current position. This type of rotation is performed when we insert a new node into the left subtree and the tree becomes unbalanced. Consider the following trees:

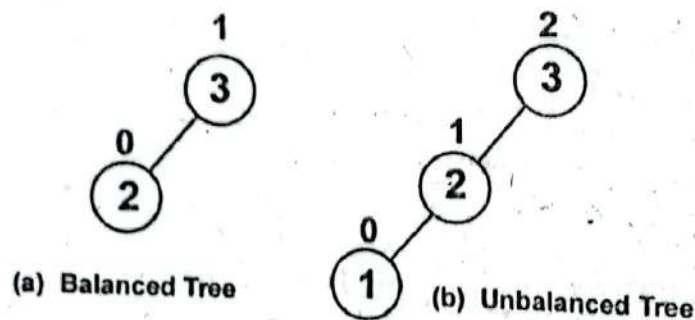


Figure 11.11: Balanced and Unbalanced Binary Trees

The tree shown in figure 11.11(a) is balanced. When a new node with value '1' is inserted into the left subtree of this tree, the tree becomes unbalanced as shown in figure 11.11(b) because the balance factor of a node with value '3' (root node) is 2. We can balance this tree by using RR Rotation (Right Rotation) which moves nodes one position right. Figure 11.12 shows the unbalanced tree before RR Rotation and balanced tree after RR Rotation.

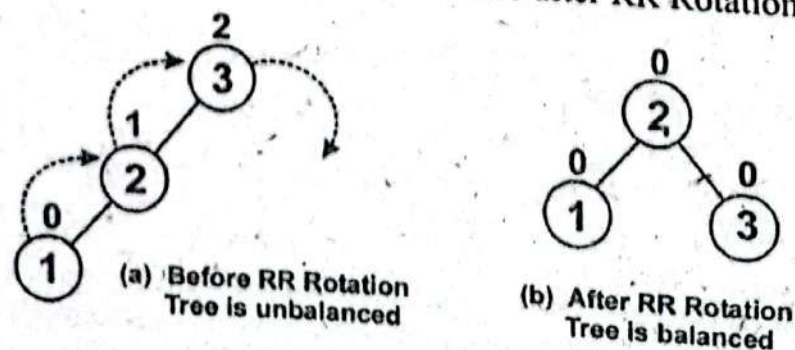


Figure 11.12: Unbalanced and Balanced Binary Trees

11.1.3.2 Double Rotation

The double rotation is further divided into two types:

- Left-Right Rotation (LR Rotation)
- Right-Left Rotation (RL Rotation)

Left-Right Rotation (LR Rotation)

The LR Rotation is a combination of a single left rotation followed by a single right rotation. In LR Rotation, first, every node moves one position to the left then one position to the right from the current position. Consider the following trees:

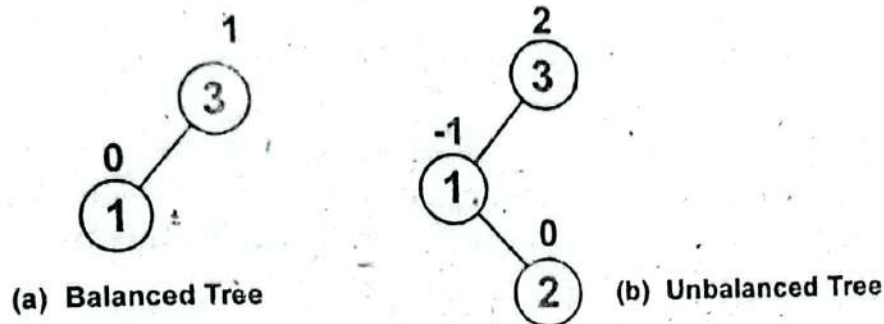


Figure 11.13: Balanced and Unbalanced Binary Trees

The tree shown in figure 11.13(a) is balanced. When a new node with value '2' is inserted into the right subtree of the left subtree, the tree becomes unbalanced as shown in figure 11.13(b) because the balance factor of a node with value '3' (root node) is 2.

We can balance this tree by using LR Rotation (Left-Right Rotation). We first perform the single left rotation (LL rotation) on the left subtree of the root node. This makes the node with value '2' as shown in figure 11.14(b). The root node with value '3' is still unbalanced. Now we perform a single right rotation (RR rotation). This makes the node with value '2', the root node of the tree. The node with value '3' becomes the right subtree, while the node with value '1' becomes the left subtree as shown in figure 11.14(c). The tree is now balanced:

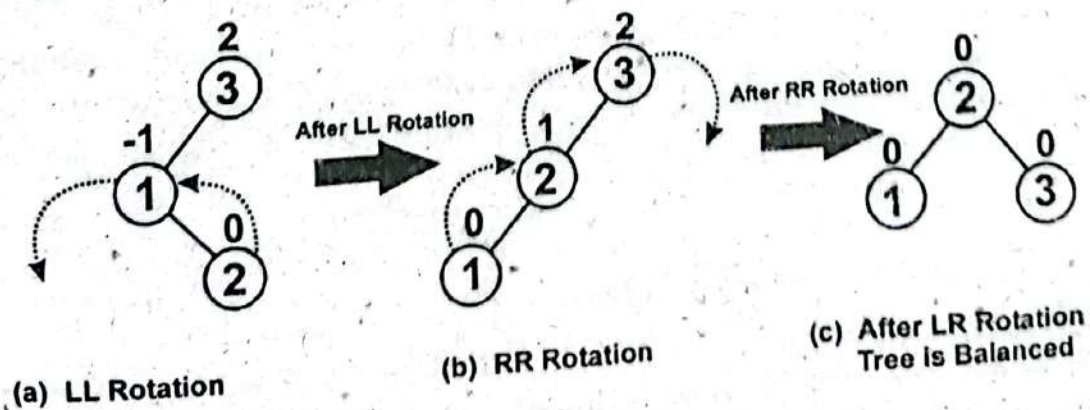


Figure 11.14: Balancing Tree by applying LL and RR rotations

ii) Right-Left Rotation (RL Rotation)

The RL Rotation is a combination of a single right rotation followed by a single left rotation. In RL Rotation, first, every node moves one position to right then one position to left from the current position. Consider the following trees:

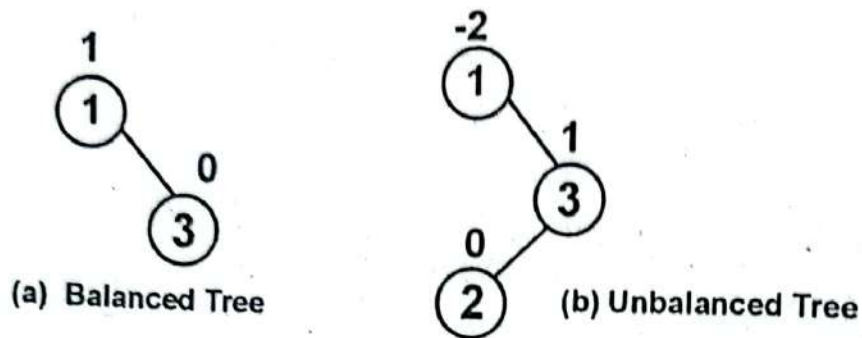


Figure 11.15: Balanced and Unbalanced Trees

The tree shown in figure 11.15 (a) is balanced. When a new node with value '2' is inserted into the left subtree of the right subtree, the tree becomes unbalanced as shown in figure 11.15(b) because the balance factor of a node with value '1' (root node) is -2.

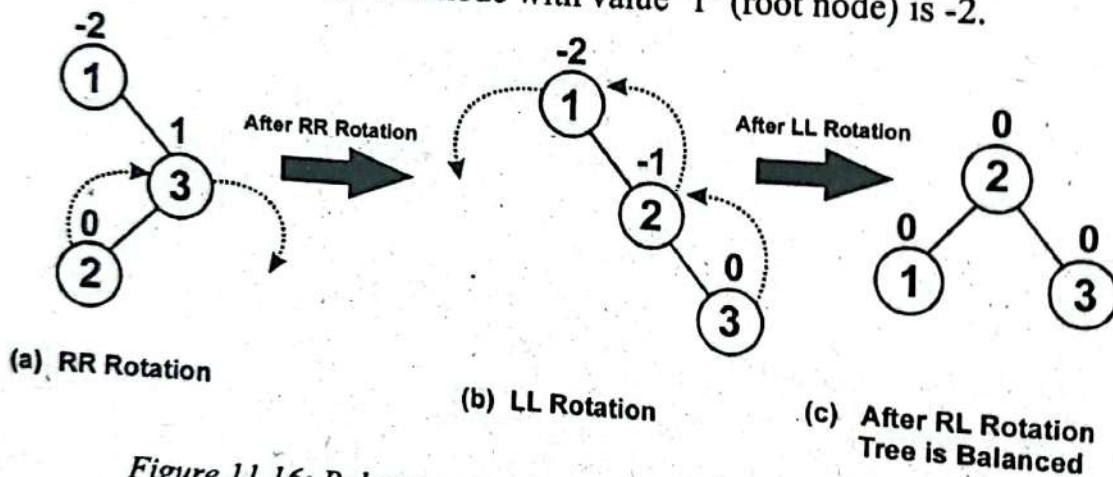


Figure 11.16: Balancing Tree by applying RR and LL rotations

We can balance this tree by using RL Rotation (Right-Left Rotation). We first perform the single right rotation (RR rotation) on the right subtree of the root node. This makes the node with value '3', the right subtree of the node with value '2' as shown in figure 11.16(b). The root node with value '1' is still unbalanced. Now we perform the single left rotation (LL rotation). This makes the node with value '2', the root node of the tree. The node with value '3' becomes the right subtree, while the node with value '1' becomes the left subtree as shown in figure 11.16(c). The tree is now balanced.

11.1.4 Insertion Operation in AVL Tree

The insertion operation in the AVL tree is performed in a similar way as that of a binary search tree. In an AVL tree, the insertion operation is performed with $O(\log n)$ time complexity. In the AVL tree, a new node is always inserted as a leaf node.

Following steps are performed for insertion operation in the AVL tree:

- 1- Insert the new node or element into the AVL tree using *binary search tree* insertion logic.
- 2- After insertion, check the *balance factor* of every node.
- 3- If the *balance factor* of every node is 0 or 1 or -1, then perform the next operation on the tree.
- 4- If the *balance factor* of any node is other than 0 or 1 or -1, then the tree is said to be unbalanced. In this case, perform a suitable rotation to make it balanced and perform the next operation on the tree.

Consider the following AVL tree:

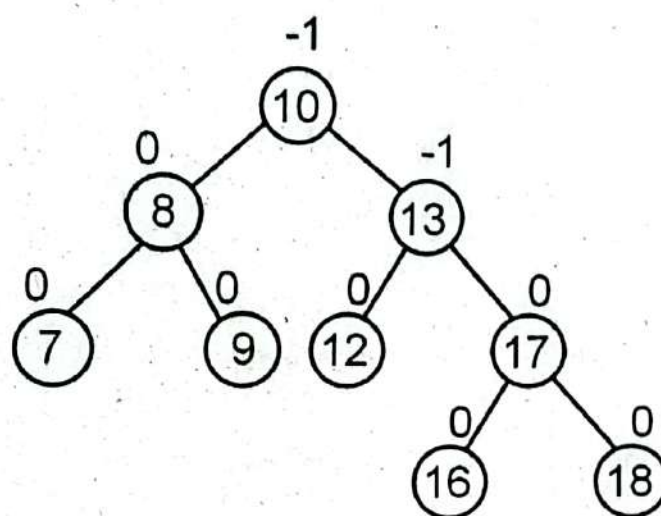
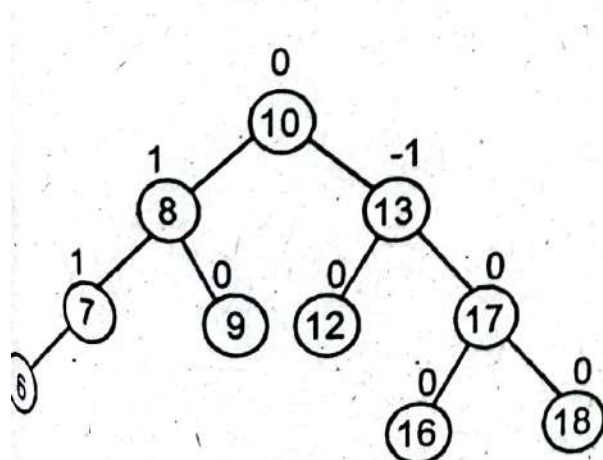
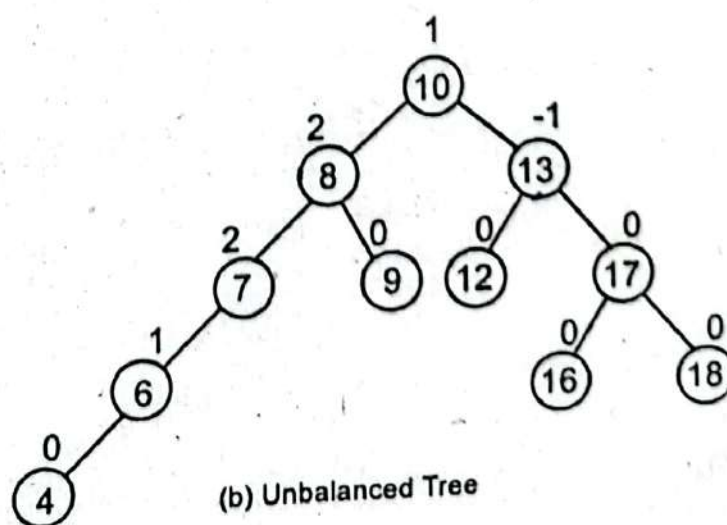


Figure 11.17: AVL Tree

Now suppose we want to insert new nodes with values 6 and 4. In the following figures, figure 11.18(a) shows the tree after inserting a new node of value 6, while figure 11.18(b) shows the tree after inserting a new node of value 4.



(a) Balanced Tree



(b) Unbalanced Tree

Figure 11.18: Trees after insertion node 6 and node 4

Note that the tree shown in figure 11.18(a) is still balanced (i.e. AVL tree) after inserting a new node of value 6. However, the tree shown in figure 11.18(b) is unbalanced because its two nodes (node with value 8 and node with value 7) have balance factor 2. It is not an AVL tree. We have to convert this unbalanced tree into an AVL tree by applying different rotation operations.

First of all, we consider the left subtree of a node with value 7, which is an unbalanced node. We apply the single right rotation (RR rotation) to this subtree (or branch of the tree). Figure 11.19 shows the balanced tree after applying RR rotation.

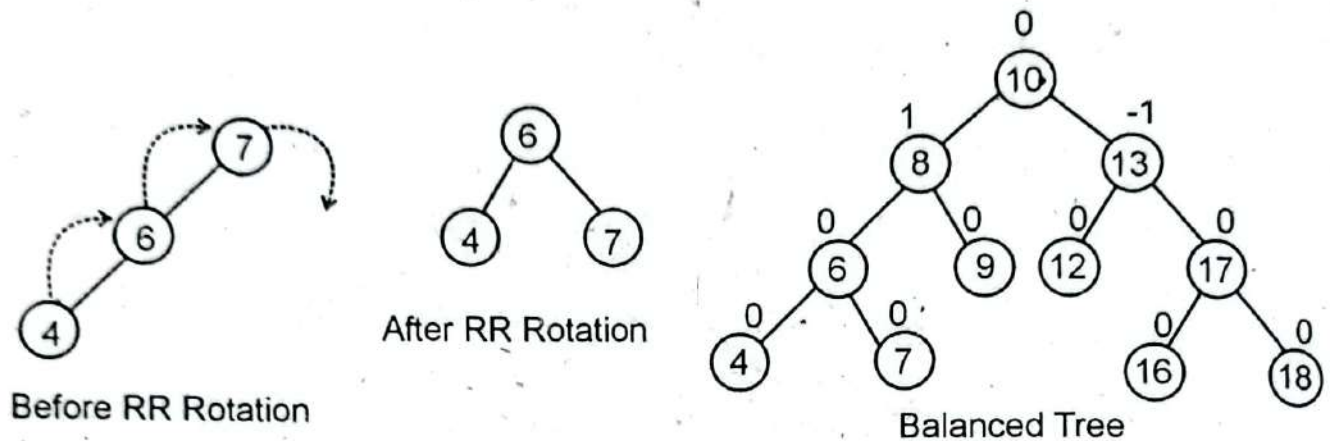


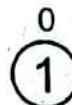
Figure 11.19: Balanced Tree after applying RR rotation

When RR Rotation is applied on the left subtree of a node with value 7, all nodes of the tree have been balanced.

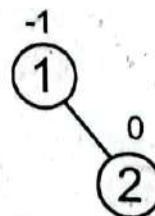
Example: Construct an AVL tree by inserting nodes with values from 1 to 6.

Perform the following steps to construct an AVL tree by inserting nodes with values from 1 to 6:

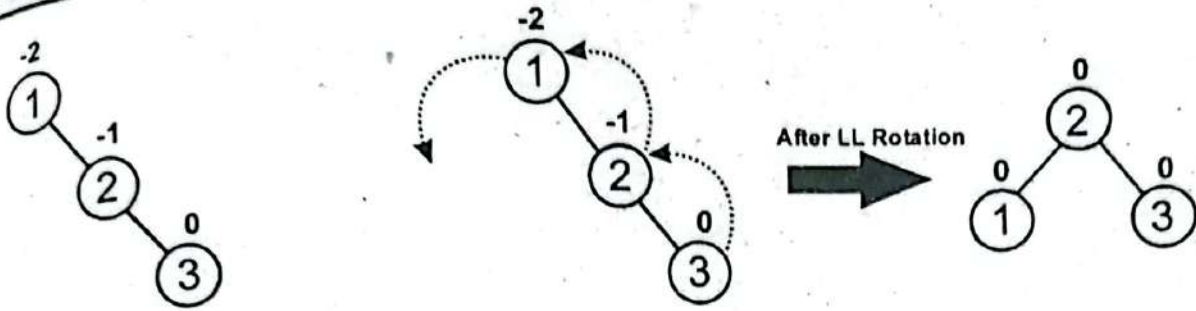
- 1- Insert node with value 1. The resulting tree is balanced.



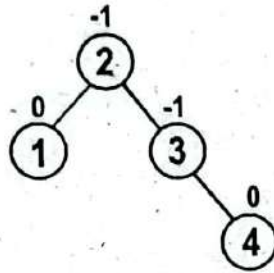
- 2- Insert node with value 2. The resulting tree is balanced.



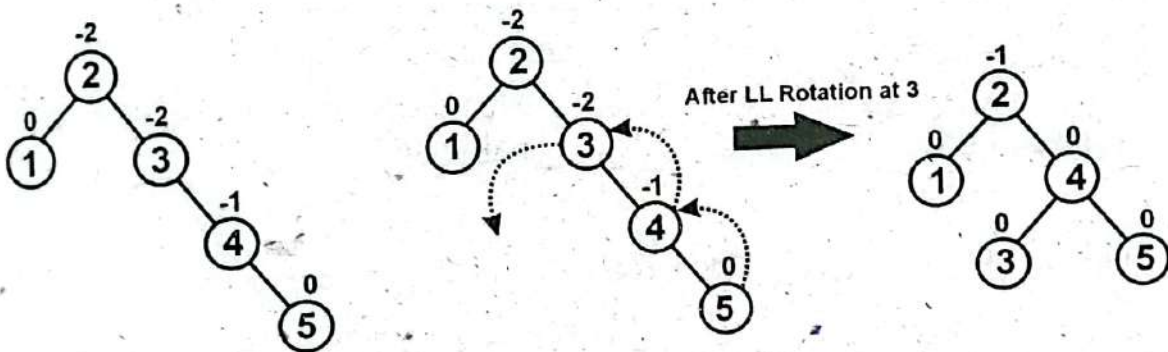
- 3- Insert node with value 3. The resulting tree is unbalanced. We can make this tree a balanced tree by applying the LL rotation operation.



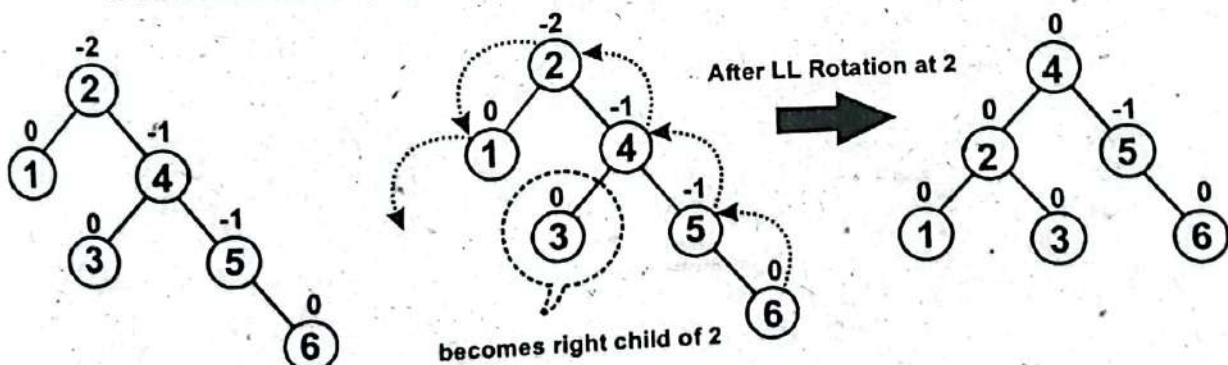
- 4- Insert node with value 4. The resulting tree is balanced.



- 5- Insert node with value 5. The resulting tree is unbalanced. We can make this tree a balanced tree by applying the LL rotation operation.



- 6- Insert node with value 6. The resulting tree is unbalanced. We can make this tree a balanced tree by applying the LL rotation operation.



11.1.5 Deletion Operation in AVL Tree

In an AVL tree, the deletion operation is similar to the deletion operation in BST but after every deletion operation, we need to check the *balance factor* condition. If the tree is balanced after deletion then go for the next operation otherwise perform the suitable rotation to make the

tree balanced. In insertion operation, only one rotation (single or double) is required to make the tree balanced. But in deletion operation, multiple rotations may be required to make the tree balanced.

Like a binary search tree, the node to be deleted in the AVL tree can be 'leaf node', 'node having only one child', and 'node having two children'. In an AVL tree, the insertion operation is performed with $O(\log n)$ time complexity.

11.1.5.1 Deleting Leaf Node

In the figures shown below, figure 11.20(a) is an AVL tree. Suppose we delete the leaf node having value 23. After deleting this node, the resulting tree becomes unbalanced as shown in figure 11.20(b). We can make this tree balanced by applying a Single Left Rotation operation as shown in figure 11.20(c). The left child node of node 30 becomes the right child of node 25.

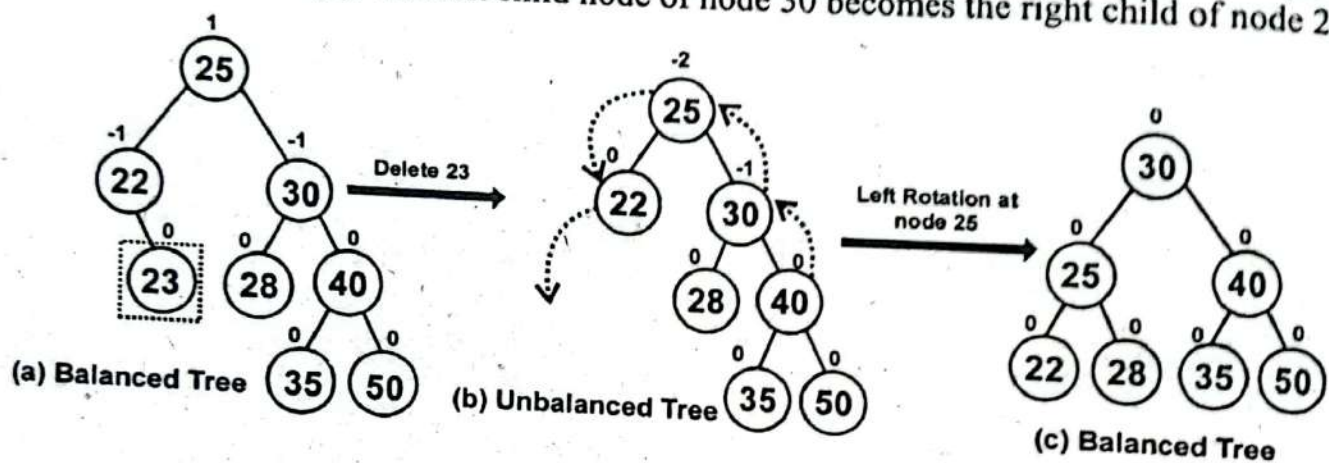


Figure 11.20: Deleting a leaf node from the AVL tree and applying left rotation

Let us take another example in which the deletion of a single leaf node requires multiple rotations to make the tree balanced. Suppose in the following AVL tree we delete the node 26.

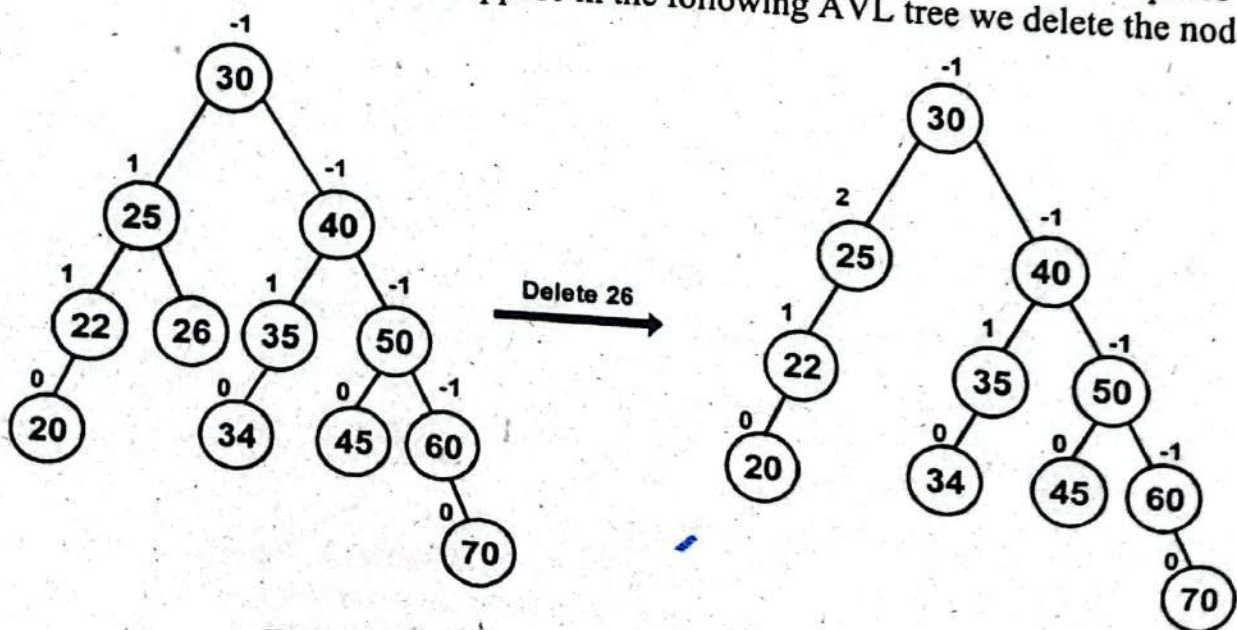


Figure 11.21: Deleting a leaf node from the AVL tree

Chap. 11 The resulting tree becomes unbalanced after deleting node 26. To make this tree balanced, we apply the Right Rotation operation at node 25. The resulting tree is still unbalanced at node 30 as shown in figure 11.22.

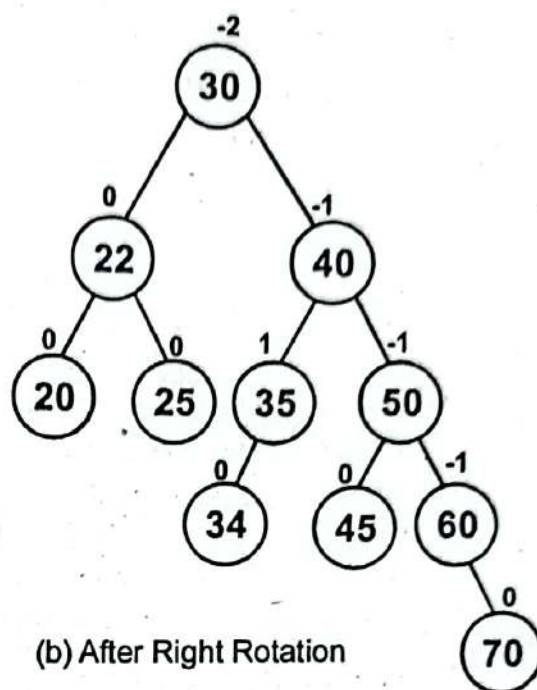
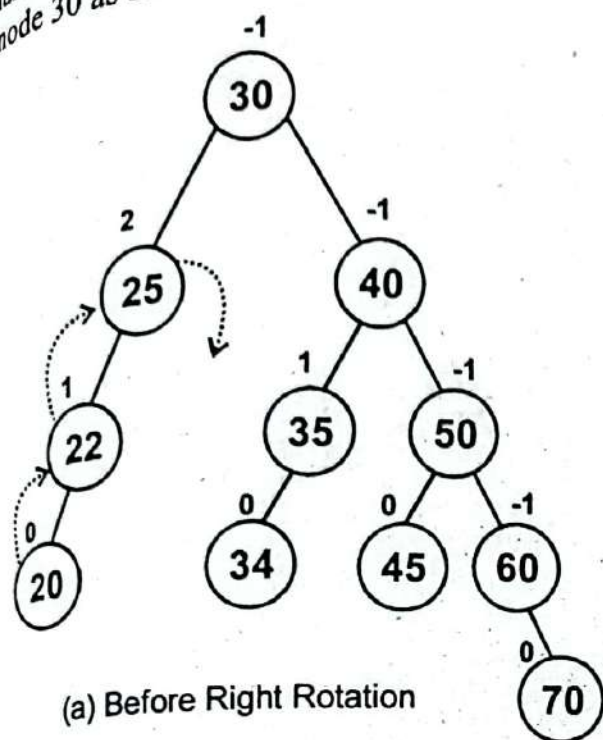


Figure 11.22: Applying right rotation at node 25

Now we apply the Left Rotation operation at node 30. The resulting tree is balanced as shown in figure 11.23.

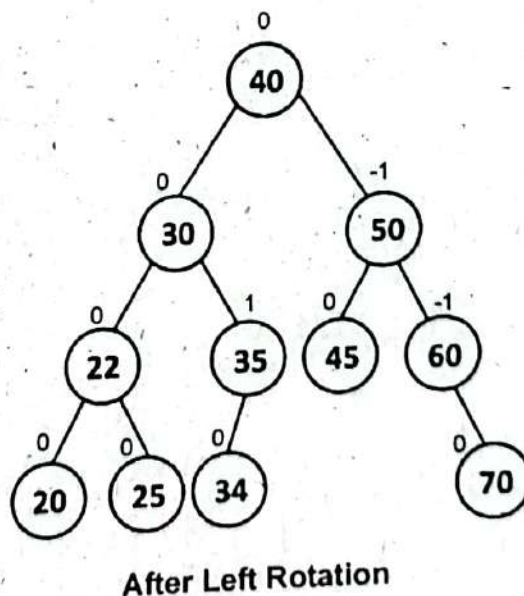
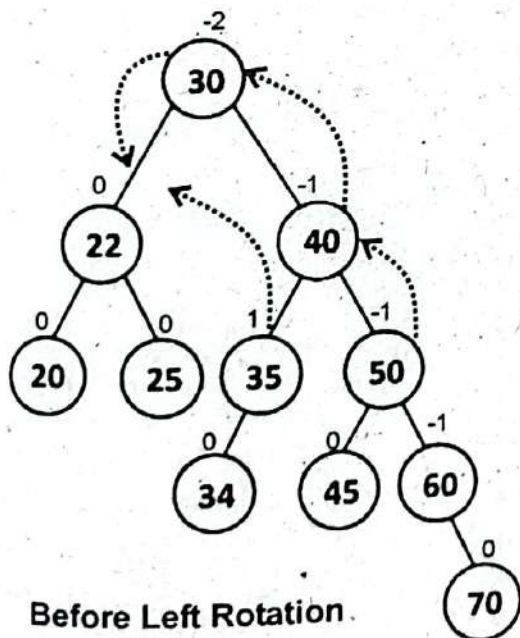


Figure 11.23: Making a balanced tree by applying left rotation at node 30

After Left Rotation, node 40 is shifted in place of node 30 and it becomes the root node, while node 30 (previous root node) is shifted left side and becomes the left subtree of node 40 (root node). The previous left subtree of node 40 becomes the right subtree of node 30.

11.1.5.2 Deleting Node Having Only One Child

Deleting a node having only one child node may unbalance multiple nodes at a time. In the figures shown below, figure 11.24(a) is an AVL tree. Suppose we delete the node 22. After deleting this node, the resulting tree becomes unbalanced as shown in figure 11.24(b). We can make this tree balanced by applying a Single Left Rotation operation as shown in figure 11.24(c).

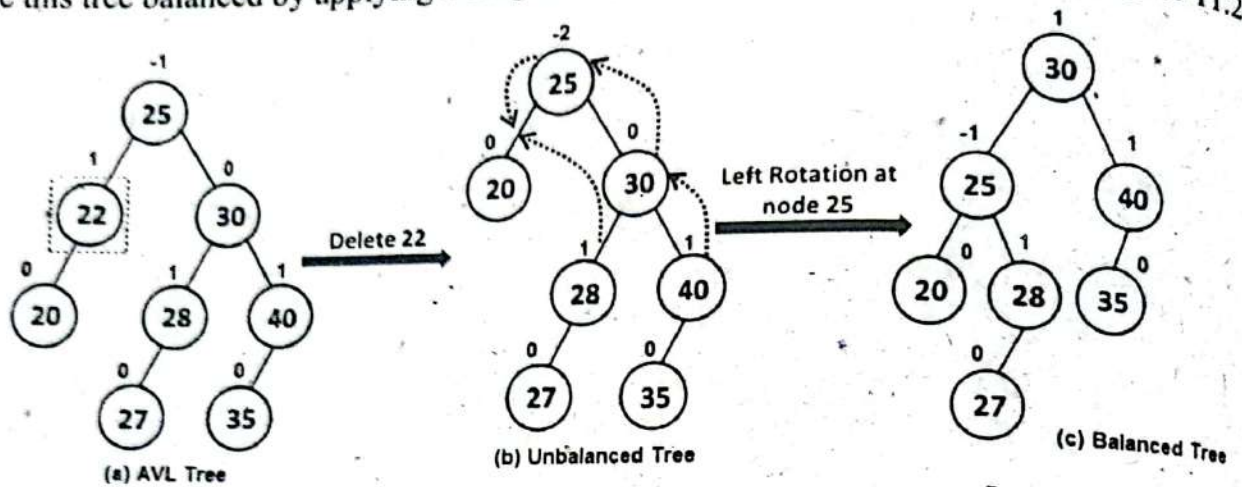


Figure 11.24: Making a balanced tree by applying left rotation at node 25

After Left Rotation, the node 30 is shifted in place of node 25 and it becomes the root node, while node 25 (previous root node) is shifted left side and becomes the left subtree of node 30 (root node). The previous left subtree of node 30 becomes the right subtree of node 25.

11.1.5.3 Deleting Node Having Two Children

Deleting a node having two children may also unbalance multiple nodes at a time. In the figures shown below, figure 11.25(a) is an AVL tree. Suppose we delete the node 68. After deleting this node, the resulting tree becomes unbalanced as shown in figure 11.25(b).

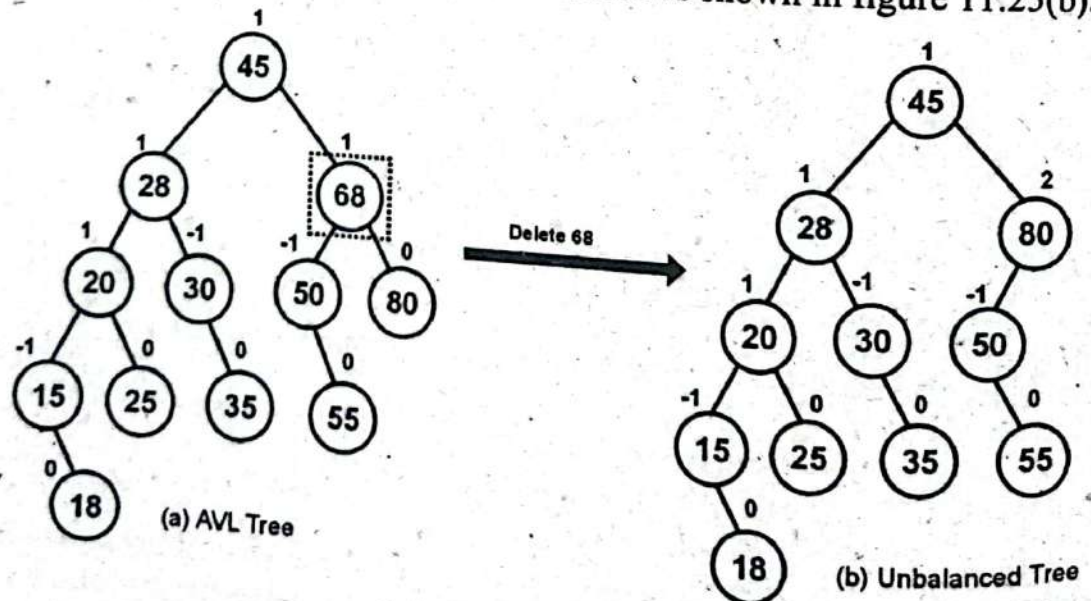


Figure 11.25: Deleting a node having two children

We can make the above-unbalanced tree as a balanced tree by applying Rotation operations. First of all, we consider the right subtree of the root node (node 45) in which node 80 is unbalanced. We apply the Left-Right Rotation (LR Rotation) on this subtree. The process of applying LR Rotation on the right subtree is shown in the following figure 11.26:

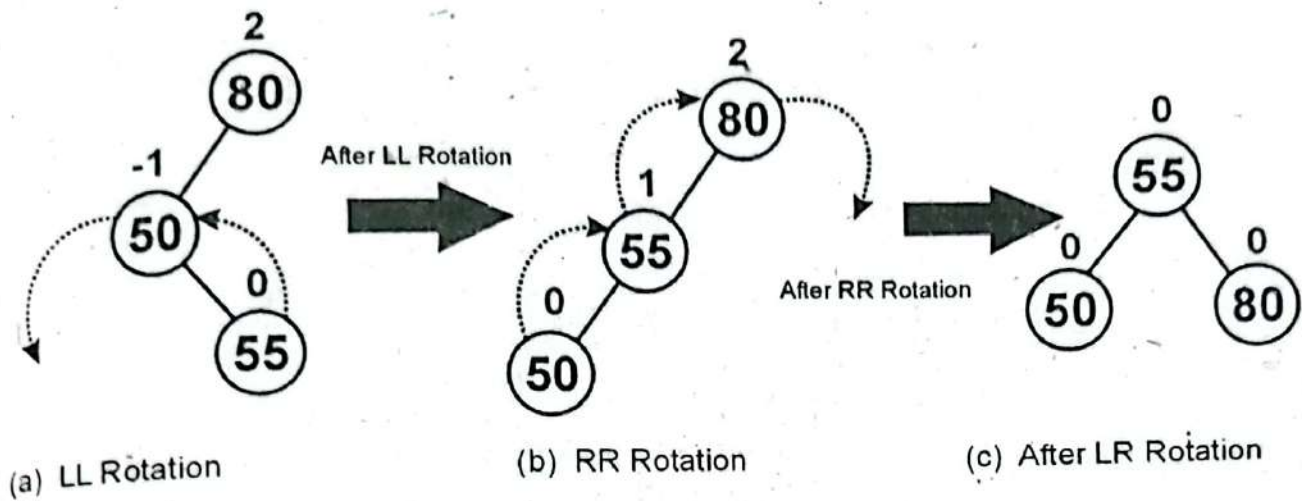


Figure 11.26: Applying LR rotation for making tree balanced.

After applying LR Rotation, the tree is still unbalanced at node 45 (root node) as shown in figure 11.27(a). Now apply the Right Rotation at node 45. After applying Right Rotation, the tree becomes balanced as shown in figure 11.27(b).

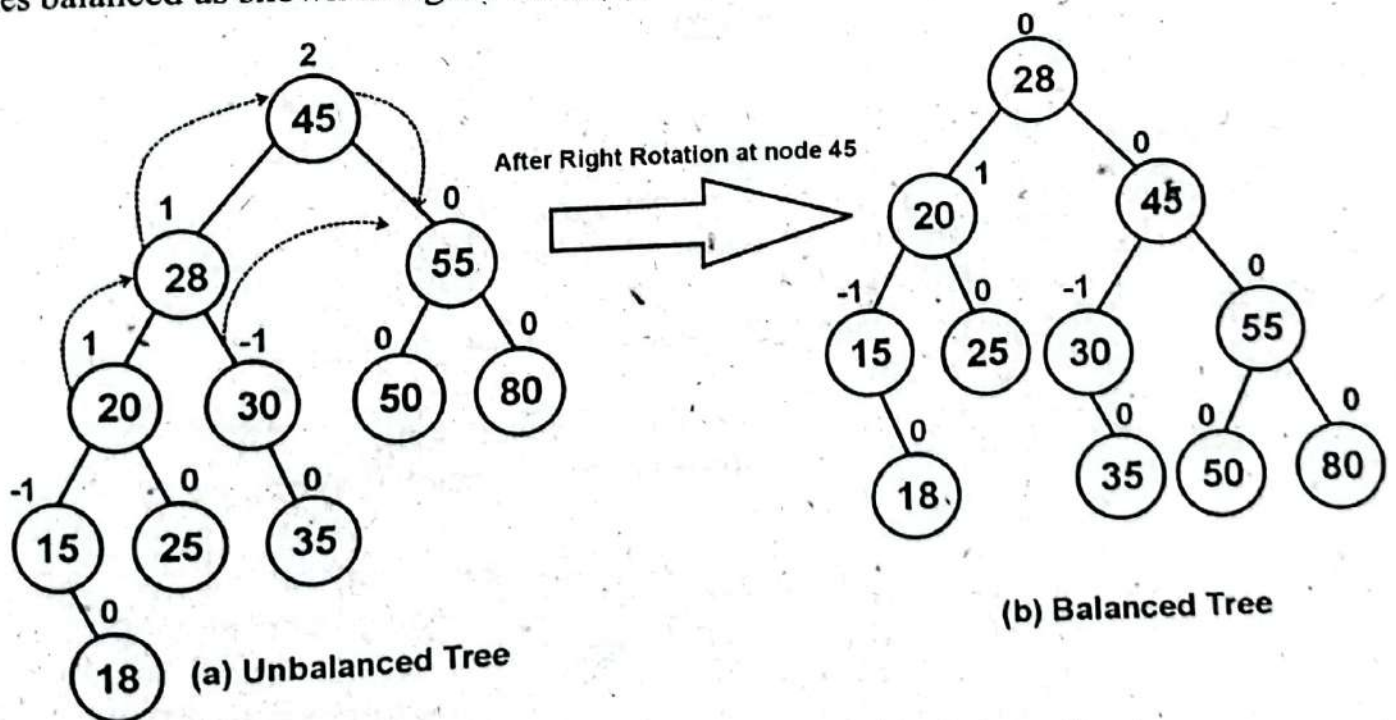


Figure 11.27: Applying LR rotation for making tree balanced

Program Code 11.1

```
// AVL Tree Implementation in C++
#include <iostream.h>
#include <conio.h>
```



```

struct node
{
    int data;
    node* left;
    node* right;
    int height;
};

class AVL
{
    node* root;

    void makeEmpty(node* t)
    {
        if(t == NULL)
            return;
        makeEmpty(t->left);
        makeEmpty(t->right);
        delete t;
    }

    // For node insertion
    node* insert(int x, node* t)
    {
        if(t == NULL)
        {
            t = new node;
            t->data = x;
            t->height = 0;
            t->left = t->right = NULL;
        }
        else if(x < t->data)
        {
            t->left = insert(x, t->left);
            if(height(t->left) - height(t->right) == 2)
            {
                if(x < t->left->data)
                    t = singleRightRotate(t);
                else
                    t = doubleRightRotate(t);
            }
        }
        else if(x > t->data)
        {
            t->right = insert(x, t->right);
            if(height(t->right) - height(t->left) == 2)
            {

```

```

        if(x > t->right->data)
            t = singleLeftRotate(t);
        else
            t = doubleLeftRotate(t);
    }
}
t->height = max(height(t->left), height(t->right))+1;
return t;
}

// Single Right Rotation
node* singleRightRotate(node* &t)
{
    node* u = t->left;
    t->left = u->right;
    u->right = t;
    t->height = max(height(t->left), height(t->right))+1;
    u->height = max(height(u->left), t->height)+1;
    return u;
}

// Single Left Rotation
node* singleLeftRotate(node* &t)
{
    node* u = t->right;
    t->right = u->left;
    u->left = t;
    t->height = max(height(t->left), height(t->right))+1;
    u->height = max(height(t->right), t->height)+1;
    return u;
}

// Double Left Rotation
node* doubleLeftRotate(node* &t)
{
    t->right = singleRightRotate(t->right);
    return singleLeftRotate(t);
}

// Double Right Rotation
node* doubleRightRotate(node* &t)
{
    t->left = singleLeftRotate(t->left);
    return singleRightRotate(t);
}

```



```

node* findMin(node* t)
{
    if(t == NULL)
        return NULL;
    else if(t->left == NULL)
        return t;
    else
        return findMin(t->left);
}

```

```

int max(int value1, int value2)
{
    return ((value1 > value2) ? value1 : value2);
}

```

```

node* findMax(node* t)
{
    if(t == NULL)
        return NULL;
    else if(t->right == NULL)
        return t;
    else
        return findMax(t->right);
}

```

```

// Deletion of node
node* remove(int x, node* t)
{
    node* temp;
    if(t == NULL) // Element not found
        return NULL;
    else if(x < t->data) // Searching for element
        t->left = remove(x, t->left);
    else if(x > t->data)
        t->right = remove(x, t->right);

    // Element found with 2 children
    else if(t->left && t->right)
    {
        temp = findMin(t->right);
        t->data = temp->data;
        t->right = remove(t->data, t->right);
    }

    // With one or zero child
    else
    {

```

```

        temp = t;
        if(t->left == NULL)
            t = t->right;
        else if(t->right == NULL)
            t = t->left;
        delete temp;
    }
    if(t == NULL)
        return t;
    t->height = max(height(t->left), height(t->right))+1;

    // If node is unbalanced
    // If left node is deleted, right case
    if(height(t->left) - height(t->right) == 2)
    {
        // right right case
        if(height(t->left->left) - height(t->left->right) == 1)
            return singleLeftRotate(t);

        // right left case
        else
            return doubleLeftRotate(t);
    }

    // If right node is deleted, left case
    else if(height(t->right) - height(t->left) == 2)
    {
        // left left case
        if(height(t->right->right) - height(t->right->left) == 1)
            return singleRightRotate(t);

        // left right case
        else
            return doubleRightRotate(t);
    }
    return t;
}

int height(node* t)
{
    return (t == NULL ? -1 : t->height);
}

int getBalance(node* t)
{
    if(t == NULL)
        return 0;
    else
        return height(t->left) - height(t->right);
}

```



```
void inorder(node* t)
{
    if(t == NULL)
        return;
    inorder(t->left);
    cout << t->data << " ";
    inorder(t->right);
}

public:
    AVL()
    {
        root = NULL;
    }

    void insert(int x)
    {
        root = insert(x, root);
    }

    void remove(int x)
    {
        root = remove(x, root);
    }

    void display()
    {
        inorder(root);
        cout << endl;
    }
};

void main(void)
{
    AVL t;
    clrscr();
    cout << "Insertion of values 20,25,15,10,30,5,35,67,45,21,10, and 89" << endl;
    t.insert(20);
    t.insert(25);
    t.insert(15);
    t.insert(10);
    t.insert(30);
    t.insert(5);
    t.insert(35);
    t.insert(67);
    t.insert(43);
    t.insert(21);
    t.insert(10);
}
```

```

t.insert(89);
cout<<"Data of tree in order after an insertion operation"<<endl;
t.display();
cout<<"Deletion of node with values 5, 35, 65, 89, 20, and 38"<<endl;
t.remove(5);
t.remove(35);
t.remove(67);
t.remove(89);
t.remove(20);
t.remove(35);
cout<<"Data of tree in order after a deletion operation"<<endl;
t.display();
getch();
}

```

Output of the program

Insertion of values 20, 25, 15, 10, 30, 5, 35, 67, 45, 21, 10, and 89

Data of tree in order after an insertion operation

5 10 15 20 21 25 30 35 43 67 89

Deletion of node with values 5, 35, 65, 89, 20, and 38

Data of tree in order after a deletion operation

10 15 21 25 30

1.6 Complexity Analysis of AVL Tree

In an AVL tree, both insertion, and deletion operations take $O(\log n)$ time, where 'n' is number of nodes in the tree earlier to the operation. Following table shows the time complexity and space complexity of insertion and deletion operations of the AVL tree:

Operation	Time Complexity		Space Complexity
	Average-Case	Worst-Case	Worst-Case
Insertion	$\Theta(\log n)$	$O(\log n)$	$O(n)$
Deletion	$\Theta(\log n)$	$O(\log n)$	$O(n)$

2 RED-BLACK TREE

Red-Black tree is a kind of self-balancing binary search tree in which every node has a color either red or black. This kind of tree was developed (invented) by a German computer scientist Rudolf Bayer in 1971.

Red-Black tree has the following properties:

- 1- Every node is either red or black.
- 2- The root node is black.
- 3- All leaves or NULL nodes are black.

- 4- If a node is red, then both its children are black.
- 5- In all the paths of the tree, there is the same number of black colored nodes.

Following tree satisfies all the properties of the red-black tree (Note that the printing of this book is black and white, therefore nodes of red color are shown as white color):

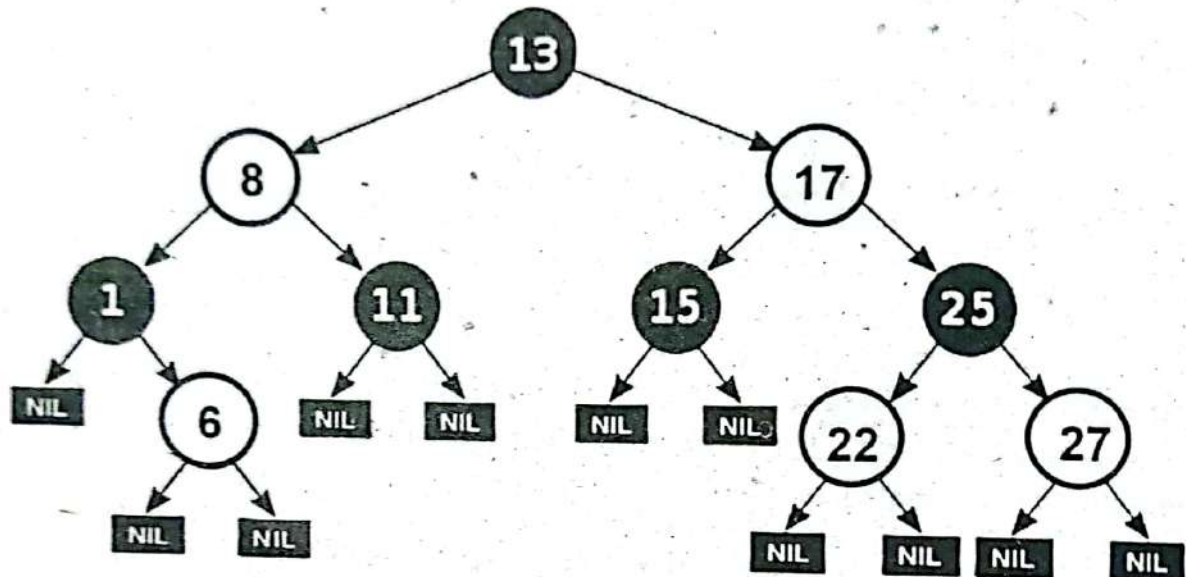


Figure 11.28: Red-Black Tree

The number of black nodes on the path from the root to the leaf is called the *black height*. The black height must be the same for all paths from the root to the leaf. The number of black nodes from the root to a node is called the *black depth of the node*.

11.2.1 Operations on Red-Black Tree

Like the AVL tree, the Red-Black tree is also a binary search tree, therefore; all the operations are performed in the same way as they are performed in a binary search tree. These operations are searching, traversing, insertion, and deletion. Searching and traversing operations do not lead to the violation of the property of the Red-Black tree. However, insertion and deletion operations can violate this property.

In the Red-Black tree, the time complexity is $O(\log n)$ for insertion and deletion operations. The space complexity is $O(n)$.

11.2.1.1 Insertion in Red-Black Tree

In a Red-Black tree, every new node must be inserted with the color red. The insertion operation in the Red-Black tree is similar to the insertion operation in the binary search tree. But it is inserted with a color property. After every insertion operation, we need to check all the properties of the Red-Black tree. If all the properties are satisfied then we go to the next operation otherwise we need to perform Recolor and rotation operations to make it a Red-Black tree.

The insertion operation in the Red-Black tree is performed using the following steps:

- 1- Check whether the tree is empty.
- 2- If the tree is Empty then insert the new node as the root node with the color black and exit from the operation.
- 3- If the tree is not empty then insert the new node as a leaf node with red color.
- 4- If the parent of the new node is black then exit from the operation.
- 5- If the parent of the new node is red then check the color of the parent node's sibling of the new node.
- 6- If it is a black or NULL node then make a suitable rotation and recolor it.
- 7- If it is a red-colored node then perform *recolor* and recheck it. Repeat the same until the tree becomes a Red-Black tree.

Example:

Construct a Red-Black tree by inserting nodes with the following sequence of values:

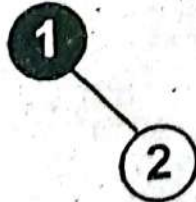
1, 2, 3, 4, 5, 6

Perform the following steps to construct the Red-Black tree:

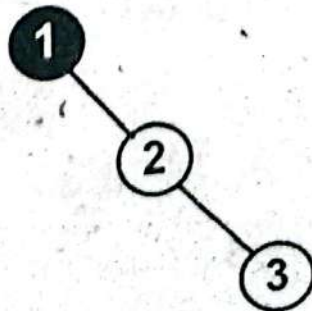
- 1- First of all insert node with value 1. The tree is empty, so insert a new node as root node with black color.



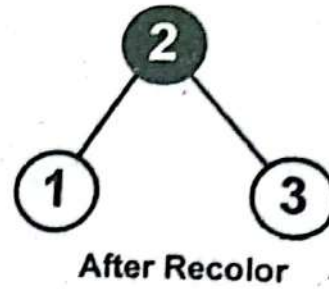
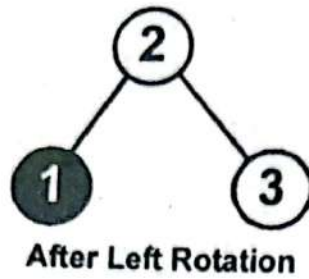
- 2- Insert node with value 2. The tree is non-empty, so insert a new node as the right child of the root node with red color.



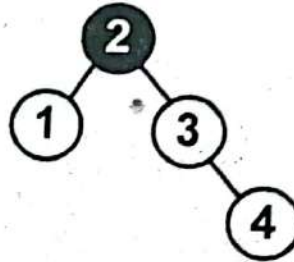
- 3- Insert node with value 3. The tree is non-empty, so insert a new node as the right child of node 2 with red color.



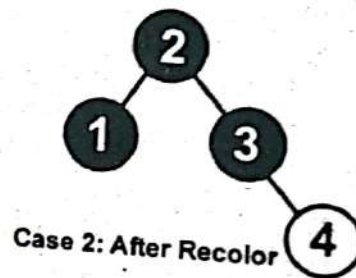
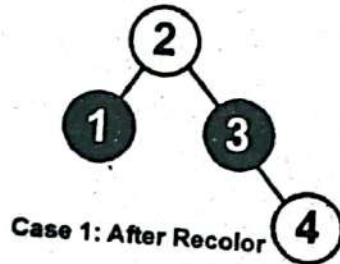
In this case, we need Left rotation and recolor the nodes.



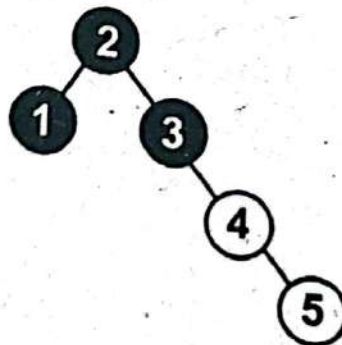
- 4- Insert node with value 4. The tree is non-empty, so insert a new node as the right child of node 3 with red color.



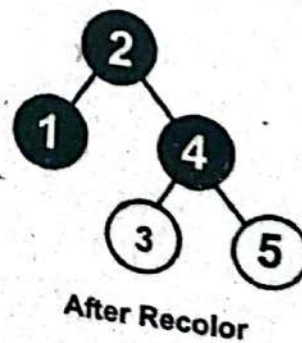
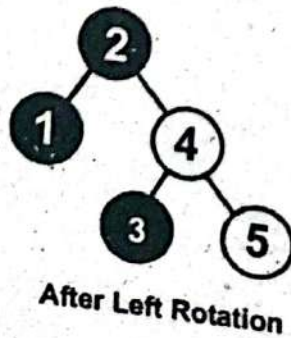
In this case, we need to recolor the nodes twice (two times).



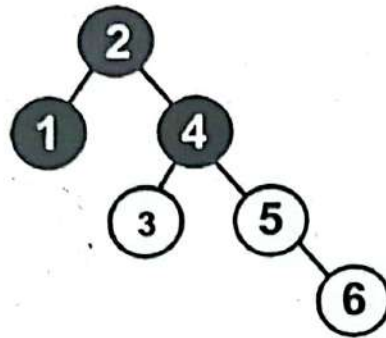
- 5- Insert node with value 5. The tree is non-empty, so insert a new node as the right child of node 4 with red color.



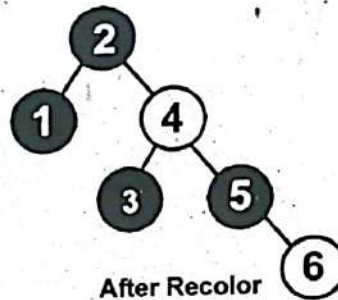
In this case, we need Left rotation and recolor the nodes.



- 6- Insert node with value 6. The tree is non-empty, so insert a new node as the right child of node 5 with red color.



In this case, we need to recolor the nodes.



11.2.1.2 Deletion in Red-Black Tree

In a Red-Black tree, the deletion operation is similar to the deletion operation in BST. But after every deletion operation, we need to check the Red-Black tree properties. If any of the property is violated then make a suitable operation like Recolor or Rotation.

11.2.1.3 Comparison of Red-Black Tree with AVL Tree

AVL tree is more balanced than the Red-Black tree, but it may cause more rotations during insertion and deletion. So if an application involves many frequent insertions and deletions, then the Red-Black tree should be preferred; otherwise, the AVL tree should be preferred.

11.3 SPLAY TREE

A splay tree is a self-adjusting (or self-balancing) binary search tree. In a splay tree, every operation on an element rearranges the tree so that the element is placed at the root of the tree. Therefore, the recently accessed elements are quick to access again. The splay tree was invented by Daniel Sleator and Robert Tarjan in 1985.

In a splay tree, every operation is performed at the root of the tree. All the normal operations on a binary search tree are combined with one basic operation, called *splaying*.

11.3.1 Splaying

The process of bringing an element of a tree to the root position by performing suitable rotation operations is known as *splaying*. It is similar to rotation operation. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. One way to do this with the basic search operation is to first perform a standard binary tree search for the element, and then use tree rotations in a specific fashion to bring the element to the top. The simplest approach to splaying is the *bottom-up* approach, which is similar to the rebalancing operations of the AVL tree.

When a particular node such as 'x' is accessed, a splay operation is performed on that node to move it to the root. To perform a splay operation we carry out a sequence of splay steps (or sub-operations), each of which moves the node closer to the root. Each particular step depends on three factors:

- Whether the node 'x' is the left or right child of its parent node such as 'p'.
- Whether parent node 'p' is the root or not.
- If parent node 'p' is not a root node, then whether parent node 'p' is the left or right child of its parent node such as grandparent 'g' of node 'x'.

There are three main types of splay steps, each of which has a left-handed and right-handed case.

1- Zig Step

Zig step is done when 'p' is the root. The tree is rotated on the edge between 'x' and 'p'. This splay step is also called the *zig rotation* which is similar to the single right rotation in AVL tree rotation. In zig rotation, every node moves one position to the right from its current position. Consider the following example:



Figure 11.29: Splaying node 3 using Zig rotation

2- Zig-Zig Step

Zig-zig step is done when 'p' is not the root and 'x' and 'p' nodes are either both right children or are both left children. The tree is rotated on the edge joining 'p' with its parent 'g', then rotated on the edge joining 'x' with 'p'. This splay step is a double zig rotation. In this rotation, every node moves two positions to the right from its current position. Consider the following example:



Figure 11.30: Splaying node 2 using Zig-Zig rotation

Zig-Zag Step

Zig-zag step is done when 'p' is not the root and 'x' is a right child and 'p' is a left child or vice versa. The tree is rotated on the edge between 'p' and 'x' and then rotated on the resulting edge between 'x' and 'g'. This splay step is a sequence of zig rotation followed by zag rotation. In these rotations, every node moves one position to the right followed by one position to the left from its current position. Consider the following example:

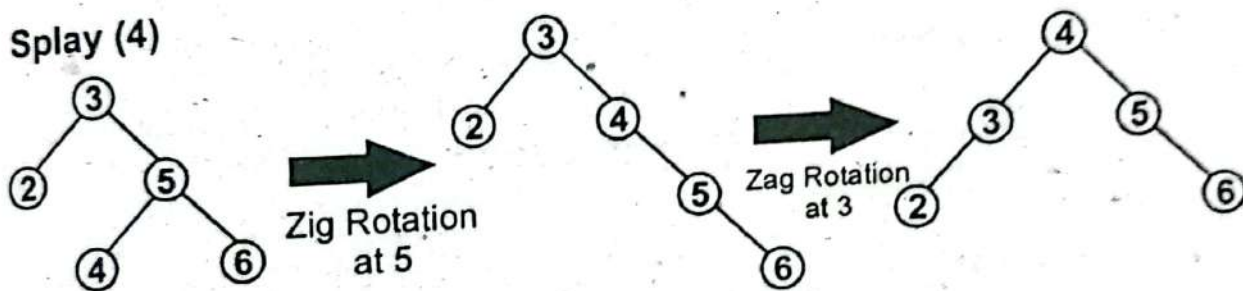


Figure 11.31: Splaying node 4 using Zig rotation and Zag rotation

Note that zag rotation is similar to the single left rotation in AVL tree rotation. In zag rotation, every node moves one position to the left from its current position. Consider the following example:

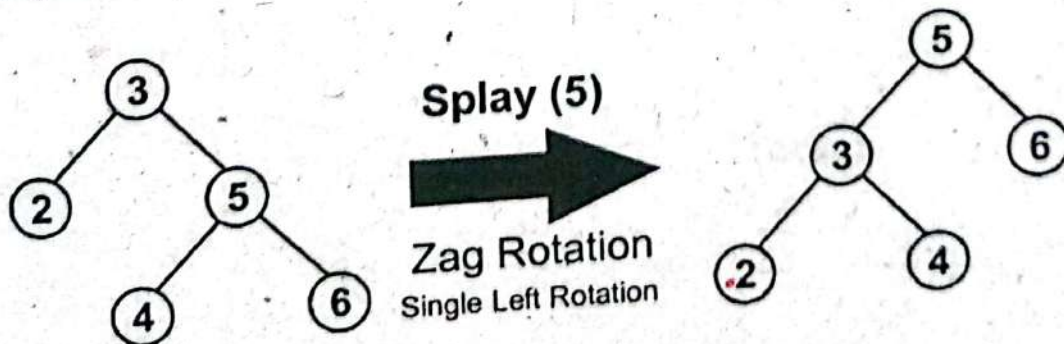
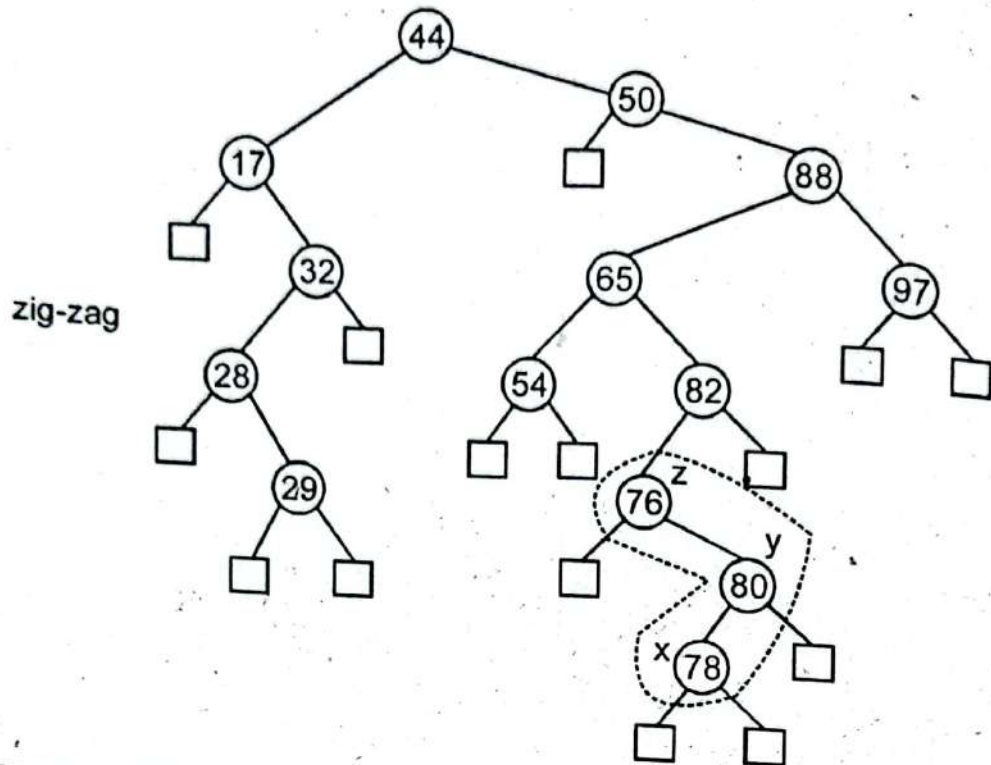


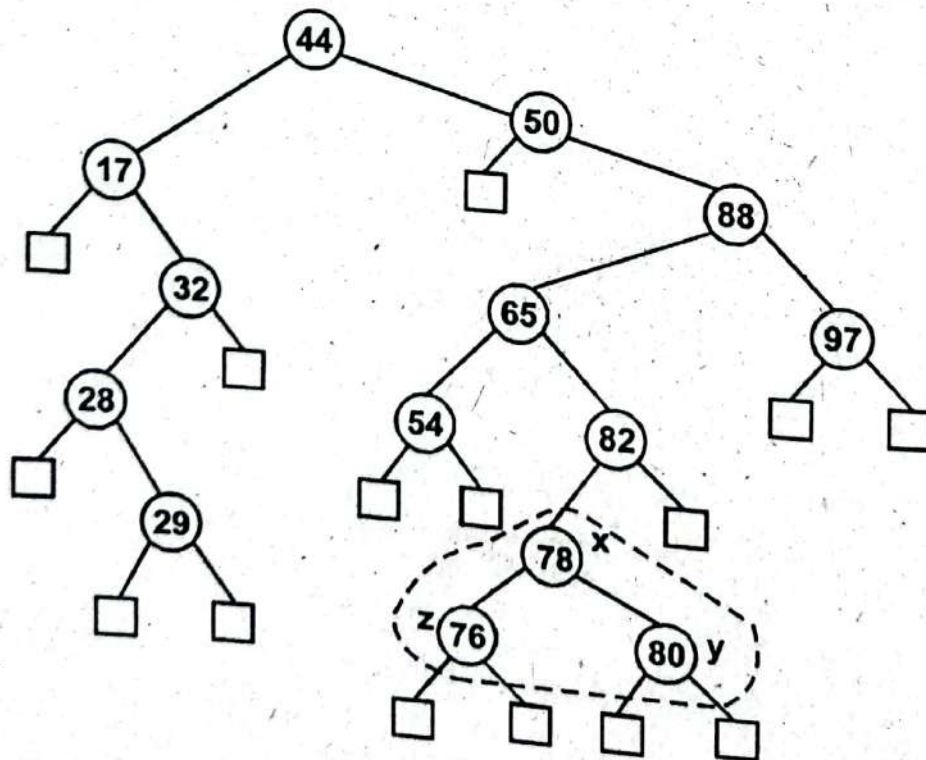
Figure 11.32: Splaying node 5 using Zag rotation

Example:

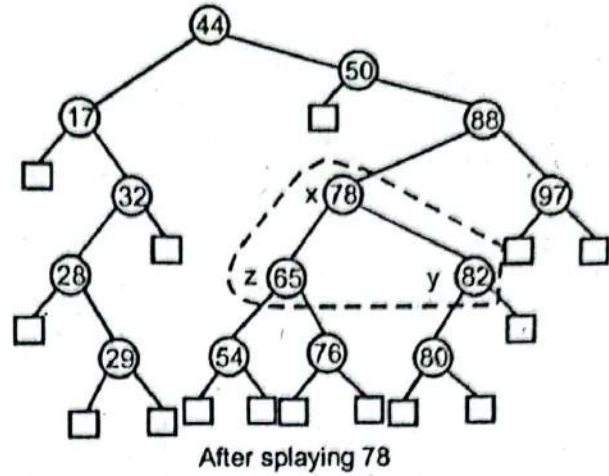
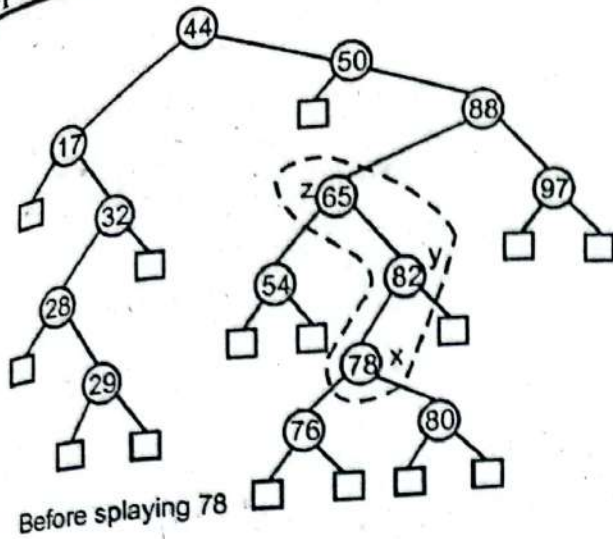
Consider the following binary search tree to explain the concept of splaying:



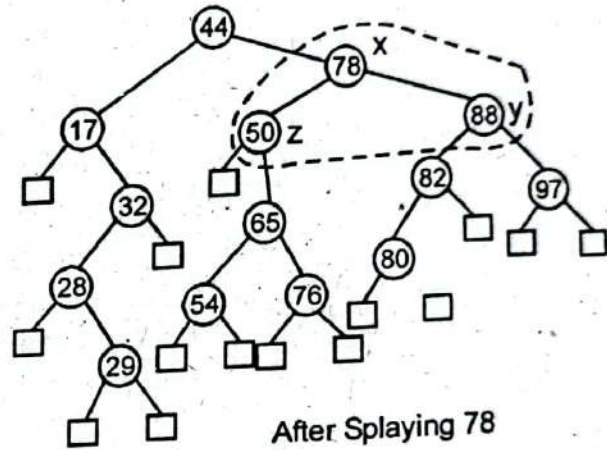
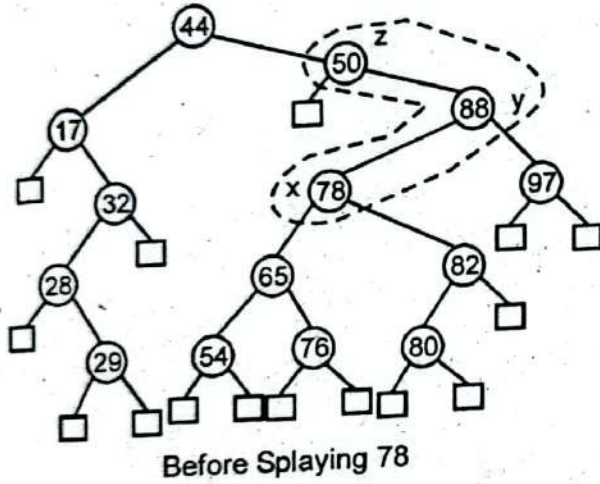
Suppose we want to splay node 78. It is denoted by 'x'. The parent node of 'x' is denoted by 'y' and the grandparent node is denoted by 'z'. We will perform the Zig-zag step because 'y' is not the root and 'x' is a right child and 'y' is a left child. The tree is rotated on the edge between 'y' and 'x' and then rotated on the resulting edge between 'x' and 'z'. The resulting tree is shown in the figure below.



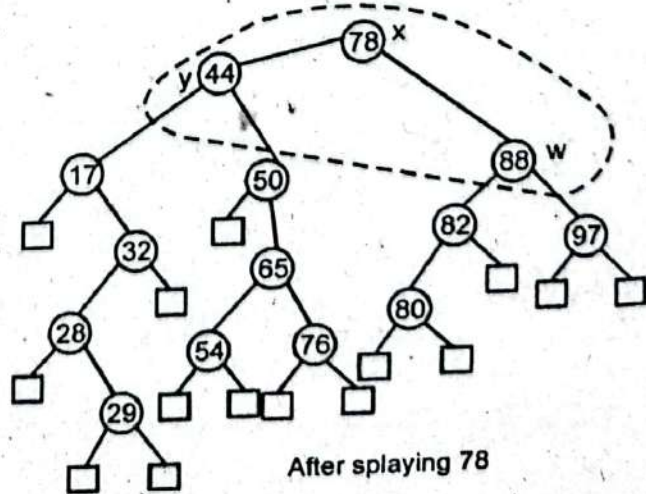
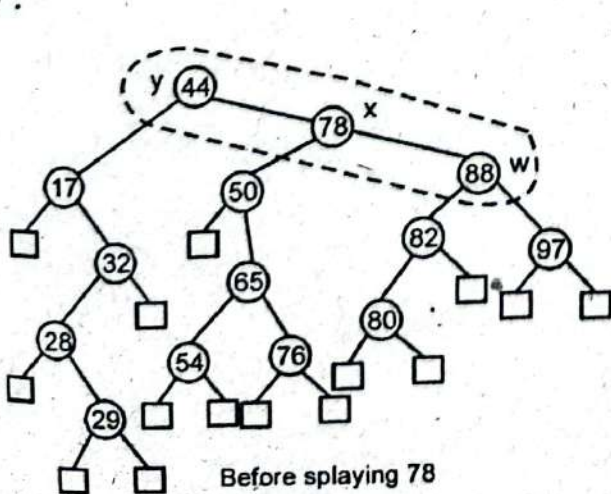
After performing the zig-zag step, node 'x' goes one position up towards the root. Now again perform the zig-zag step to splay the node 78.



Node 'x' goes further one position up towards the root. Now again perform the zig-zag step to splay the node 78.



Node 'x' goes further one position up towards the root. Now perform the zig step to splay the node 78 because now node 'y' is the root node. The tree is rotated on the edge between 'x' and 'y'.



After the completion of the splaying process, node 78 is shifted at the position of the root.

11.3.2 Join Operation In Splay Tree

Two splay trees can be joined into one if all items in the first tree are smaller than all items in the second tree. Join operation is mostly used after applying the deletion operation on the splay tree. Suppose two splay trees are T1 and T2. The following steps can be performed to join them to a single tree:

- Splay the largest item of T1 to the root. Now, this item is in the root of T1 and has a null right child.
- Set T2 as the right child of T1.

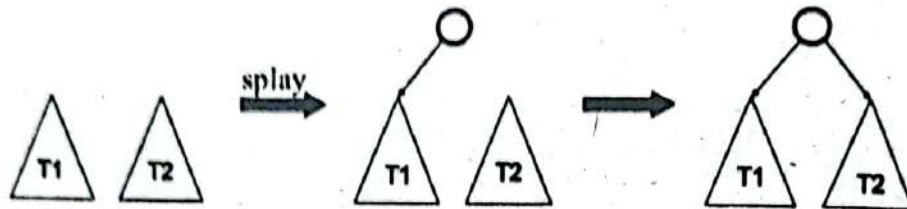


Figure 11.33: Join Operation in Splay Tree

11.3.3 Split Operation In Splay Tree

Given an item 'x', we can split a splay tree into two new trees: one containing all items smaller than or equal to 'x' and the other containing all items larger than 'x'. Suppose T is a splay tree with left subtree 'L' and right subtree 'R'. We want to split T at a key value 'x' into two trees T1 and T2. The following steps can be performed to split this tree into new trees at item 'x':

- Splay 'x' to the root. So the resulting tree contains all items smaller than or equal to 'x' to its left and all items larger than 'x' to its right.
- Split the tree by un-linking the right subtree. The tree T will be divided into two trees say T1 and T2.

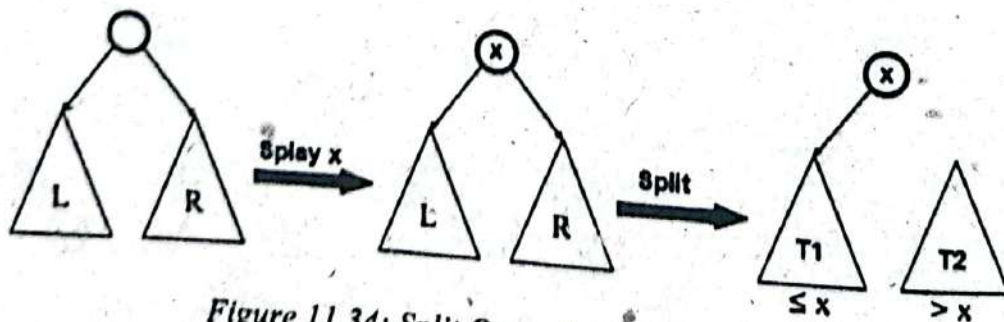


Figure 11.34: Split Operation in Splay Tree

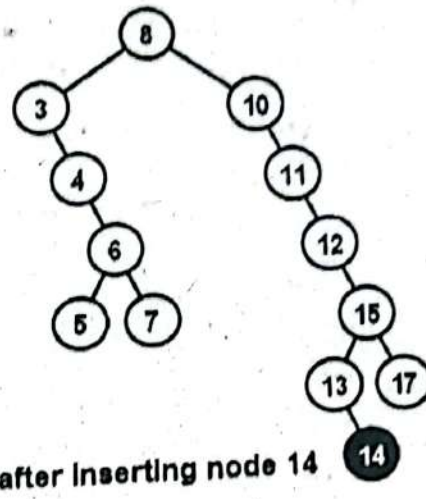
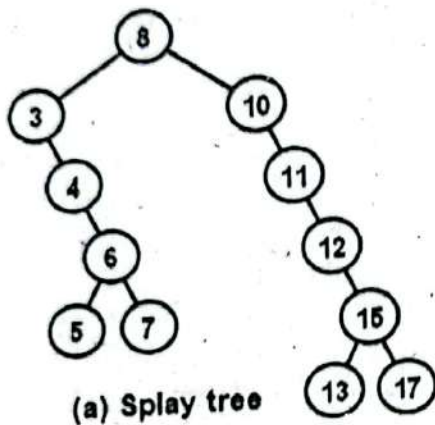
11.3.4 Insertion Operation In Splay Tree

Like insertion operation in an AVL tree, the insertion operation in the splay tree is also performed with $O(\log n)$ time complexity. The insertion operation in the splay tree is performed using the following steps:

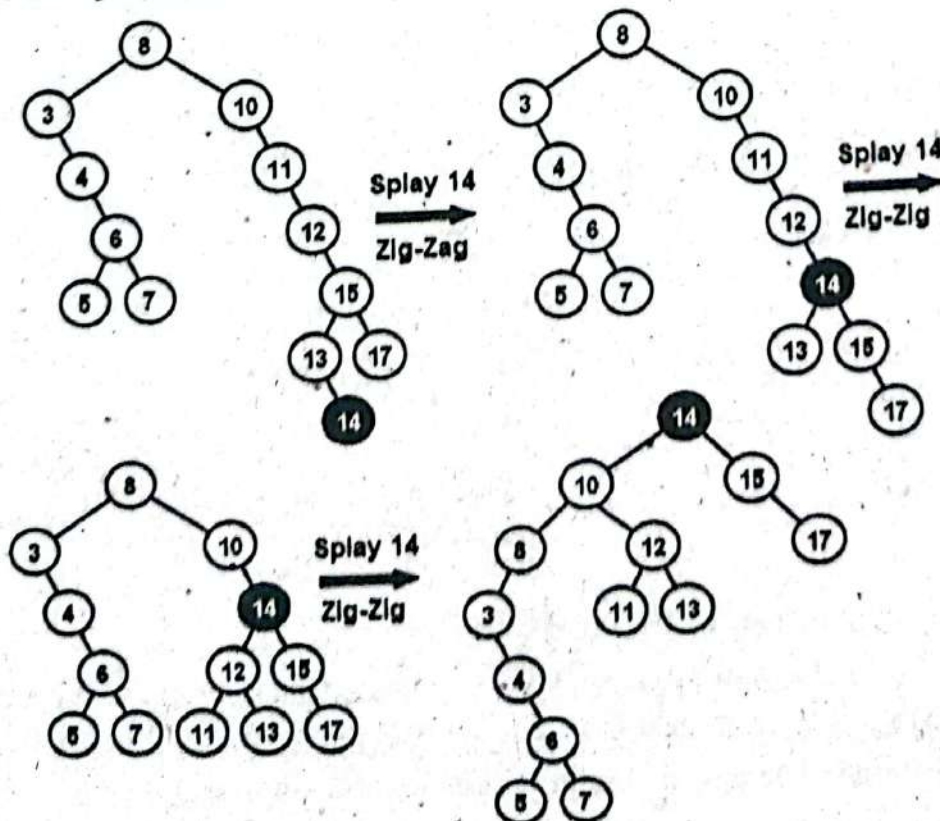
- 1- Check whether the tree is empty.
- 2- If the tree is empty, then insert the new node as the root node and terminate the operation.
- 3- If the tree is not empty, then insert the new node as a leaf node using binary search tree insertion logic.
- 4- After insertion, splay the new node.

Example:

Consider the following splay tree as shown in figure (a). We insert a new node with value 14 into this tree as shown in figure (b).



Now we splay the new node 14. The following figures show the splaying of node 14. After completion of the splaying process, node 14 will be shifted to root.

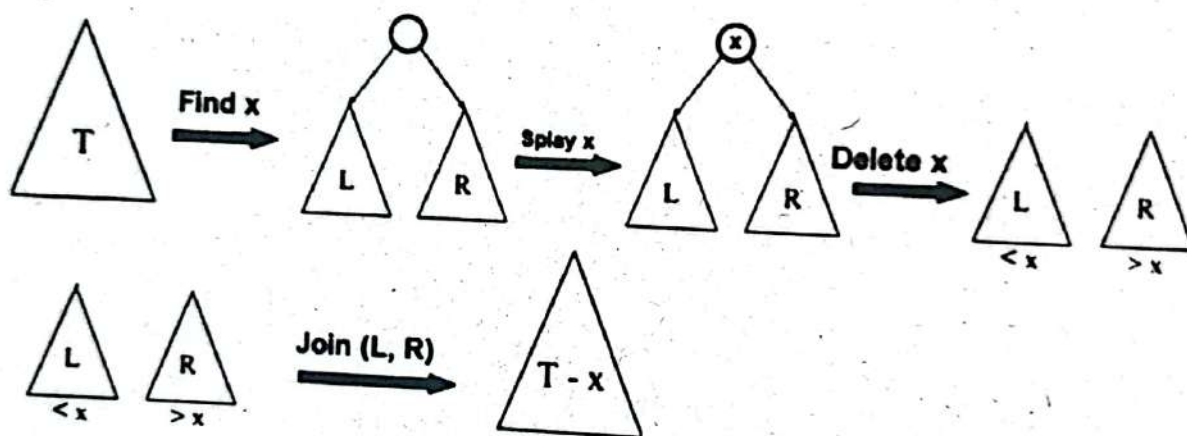


11.3.5 Deletion Operation in Splay Tree

In a splay tree, the deletion operation is similar to the deletion operation in a binary search tree. Like deletion operation in an AVL tree, the deletion operation in the splay tree is also performed with $O(\log n)$ time complexity.

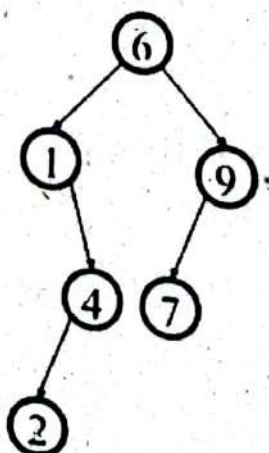
The deletion operation in the splay tree is performed using the following steps:

- 1- Find the item in the tree which is to be deleted. If found, then splay the item to the root; otherwise terminate the deletion operation.
- 2- Delete the item. The tree is divided into two subtrees such as L and R.
- 3- Join the two subtrees by performing the following steps:
 - Splay the largest item of subtree L to the root. Now, this item is in the root of L and has a null right child.
 - Set subtree R as the right child of subtree L.

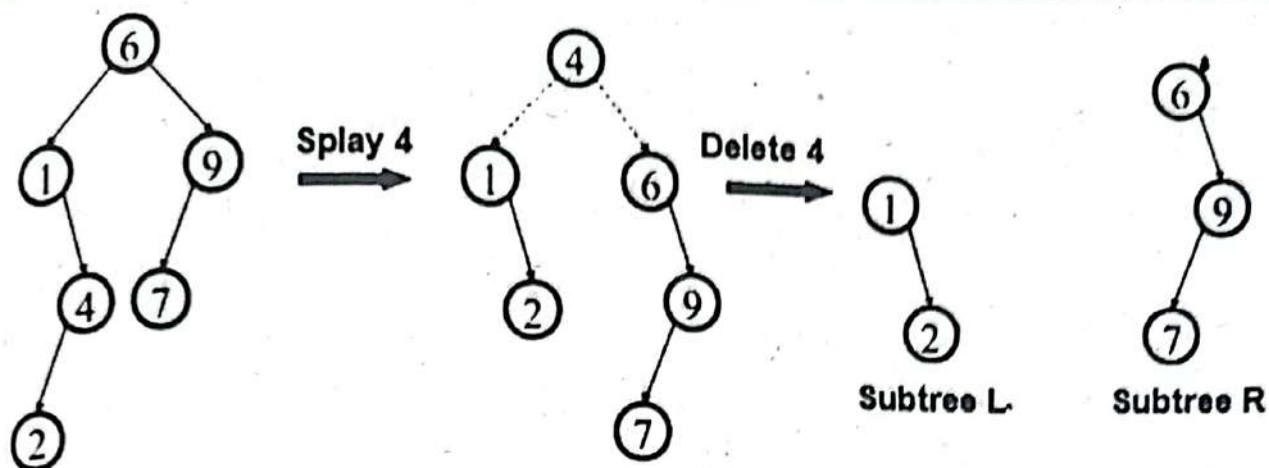


Example:

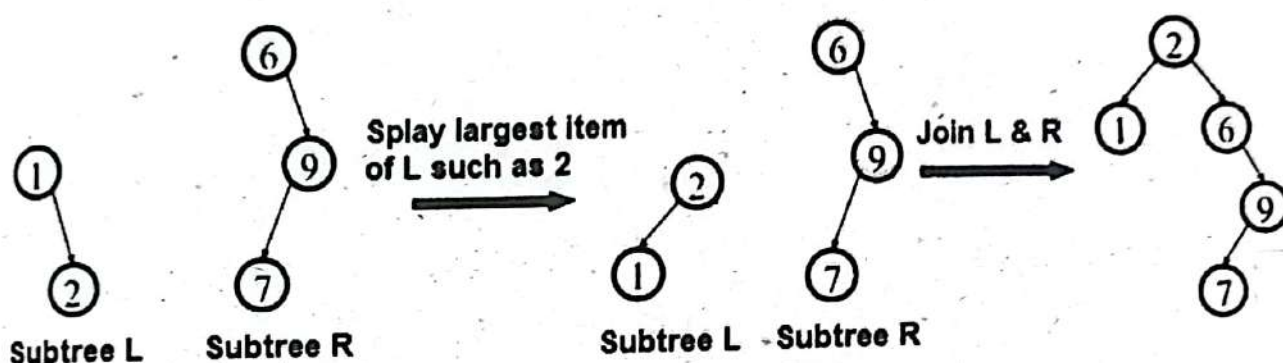
Consider the following splay tree:



Suppose we want to delete node 4, which is present in the tree. Splay this node to the root and then delete this node. The tree will be divided into two subtrees L and R.



Now, splay the largest item of subtree L to the root and then join the subtrees L and R. The resulting tree is shown in the following figure:



Program Code 11.2

// C++ Program to Implement Splay Tree

```
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
```

```
struct splay
{
```

```
    int key;
    splay* lchild;
    splay* rchild;
```

```
};
```

```
class SplayTree
```

```
{
```

```
public:
    SplayTree()
    {
    }
```

```
// RR(Y rotates to the right)
```



```

splay* RR_Rotate(splay* k2)
{
    splay* k1 = k2->lchild;
    k2->lchild = k1->rchild;
    k1->rchild = k2;
    return k1;
}

// LL(Y rotates to the left)
splay* LL_Rotate(splay* k2)
{
    splay* k1 = k2->rchild;
    k2->rchild = k1->lchild;
    k1->lchild = k2;
    return k1;
}

// An implementation of top-down splay tree
splay* Splay(int key, splay* root)
{
    if (!root)
        return NULL;
    splay header;

    // header.rchild points to L tree;
    // header.lchild points to R Tree
    header.lchild = header.rchild = NULL;
    splay* LeftTreeMax = &header;
    splay* RightTreeMin = &header;
    while (1)
    {
        if (key < root->key)
        {
            if (!root->lchild)
                break;
            if (key < root->lchild->key)
            {
                root = RR_Rotate(root);

                // only zig-zig mode need to rotate once,
                if (!root->lchild)
                    break;
            }

            //Link to R Tree

```

```

        RightTreeMin->lchild = root;
        RightTreeMin = RightTreeMin->lchild;
        root = root->lchild;
        RightTreeMin->lchild = NULL;
    }
    else if (key > root->key)
    {
        if (lroot->rchild)
            break;
        if (key > root->rchild->key)
        {
            root = LL_Rotate(root);

            // only zag-zag mode need to rotate once,
            if (lroot->rchild)
                break;
        }

        // Link to L Tree
        LeftTreeMax->rchild = root;
        LeftTreeMax = LeftTreeMax->rchild;
        root = root->rchild;
        LeftTreeMax->rchild = NULL;
    }
    else
        break;
}

// assemble L Tree, Middle Tree and R tree
LeftTreeMax->rchild = root->lchild;
RightTreeMin->lchild = root->rchild;
root->lchild = header.rchild;
root->rchild = header.lchild;
return root;
}

splay* New_Node(int key)
{
    splay* p_node = new splay;
    if (!p_node)
    {
        fprintf(stderr, "Out of memory!\n");
        exit(1);
    }
    p_node->key = key;
    p_node->lchild = p_node->rchild = NULL;
    return p_node;
}

```



```

    }

    splay* Insert(int key, splay* root)
    {
        static splay* p_node = NULL;
        if (!p_node)
            p_node = New_Node(key);
        else
            p_node->key = key;
        if (!root)
        {
            root = p_node;
            p_node = NULL;
            return root;
        }
        root = Splay(key, root);

        /* This is BST that, all keys <= root->key is in root->lchild,
           all keys >root->key is in root->rchild. */
        if (key < root->key)
        {
            p_node->lchild = root->lchild;
            p_node->rchild = root;
            root->lchild = NULL;
            root = p_node;
        }
        else if (key > root->key)
        {
            p_node->rchild = root->rchild;
            p_node->lchild = root;
            root->rchild = NULL;
            root = p_node;
        }
        else
            return root;
        p_node = NULL;
        return root;
    }

    splay* Delete(int key, splay* root)
    {
        splay* temp;
        if (!root)
            return NULL;
        root = Splay(key, root);
        if (key != root->key)

```

```

        return root;
    else
    {
        if (lroot->lchild)
        {
            temp = root;
            root = root->rchild;
        }
        else
        {
            temp = root;

```

/*Note: Since key == root->key,
so after Splay(key, root->lchild),
the tree we get will have no right child tree.*/

```

        root = Splay(key, root->lchild);
        root->rchild = temp->rchild;

```

```

    }
    free(temp);
    return root;
}

```

```

splay* Search(int key, splay* root)
{
    return Splay(key, root);
}

```

```

void InOrder(splay* root)
{

```

```

    if (root)
    {

```

```

        InOrder(root->lchild);
        cout<< "key: " << root->key;
        if(root->lchild)
            cout<< " | left child: " << root->lchild->key;
        if(root->rchild)
            cout << " | right child: " << root->rchild->key;
        cout<< "\n";
        InOrder(root->rchild);
    }
}

```

```

};
void main(void)

```



```

{
    SplayTree st;
    int vector[10] = {9,8,7,6,5,4,3,2,1,0};
    splay *root;
    root = NULL;
    const int length = 10;
    int i;
    for(i = 0; i < length; i++)
        root = st.Insert(vector[i], root);
    clrscr();
    cout<<"\nInOrder: \n";
    st.InOrder(root);
    int input, choice;
    while(1)
    {
        cout<<"\nSplay Tree Operations\n";
        cout<<"1. Insert "<<endl;
        cout<<"2. Delete"<<endl;
        cout<<"3. Search"<<endl;
        cout<<"4. Exit"<<endl;
        cout<<"Enter your choice: ";
        cin>>choice;
        switch(choice)
        {
            case 1:
                cout<<"Enter value to be inserted: ";
                cin>>input;
                root = st.Insert(input, root);
                cout<<"\nAfter Insert: "<<input<<endl;
                st.InOrder(root);
                break;
            case 2:
                cout<<"Enter value to be deleted: ";
                cin>>input;
                root = st.Delete(input, root);
                cout<<"\nAfter Delete: "<<input<<endl;
                st.InOrder(root);
                break;
            case 3:
                cout<<"Enter value to be searched: ";
                cin>>input;
                root = st.Search(input, root);
                cout<<"\nAfter Search "<<input<<endl;
                st.InOrder(root);
                break;
        }
    }
}

```

case 4:

exit(1);

default:

cout<<"\nInvalid type! \n";

}

}

cout<<"\n";

getch();

Output of the program:

InOrder:

key: 0 | right child: 1

key: 1 | right child: 2

key: 2 | right child: 3

key: 3 | right child: 4

key: 4 | right child: 5

key: 5 | right child: 6

key: 6 | right child: 7

key: 7 | right child: 8

key: 8 | right child: 9

key: 9

Splay Tree Operations

1. Insert

2. Delete

3. Search

4. Exit

Enter your choice:

11.4 B-Trees

The binary search tree, AVL tree, Red-Black tree, etc. are binary trees. In these trees, every node can have only one key (value) and a maximum of two children. There is another type of popular search tree that is not binary. This tree is called *B-tree*. In this type of search tree, every node can have multiple values (keys) and more than two children. It is also a self-balancing tree-like AVL tree. B-tree was developed by Bayer and McCreight in 1971 with the name of *height-balanced m-way search tree*. Later it was named as *B-tree*. This data structure is commonly used in databases and file systems.

The structure of the *B-tree* is very simple. It has a root node, interior nodes, and leaf nodes. Interior nodes are all nodes except for the root node and leaf nodes. Actual data is stored at leaf nodes, while the interior nodes contain pointers $P_1, P_2, \dots, P_M$ to the children, and values $k_1, k_2, \dots, k_{M-1}$, representing the smallest key found in the subtrees $P_2, P_3, \dots, P_M$, respectively. Some of these pointers might be *NULL*, and the corresponding k_i would then be undefined. For every node, all the keys in subtree P_1 are smaller than the keys in subtree P_2 , and so on. Each interior

node's keys act as separation values that divide subtrees. For example, if an interior node has 3 child nodes (or subtrees) then it must have 2 keys: let's 'x' and 'y'. All values in the leftmost subtree will be less than 'x', all values in the middle subtree will be between 'x' and 'y', and all values in the rightmost subtree will be greater than 'y'.

The number of keys in a node (non-leaf node) and the number of children for a node depend on the order (the maximum number of children for each node) of the *B-tree*. Every *B-tree* has an order. In a *B-tree* of order M , each non-leaf node contains a maximum of $M-1$ keys and maximum M children. For example, in a *B-tree* of order 4, each non-leaf node contains a maximum of 3 keys and a maximum of 4 children. A *B-tree* of order 4 is known as a 2-3-4 tree. Similarly, a *B-tree* of order 3 is known as a 2-3 tree. A *B-tree* of order 5 is shown in figure 11.35.

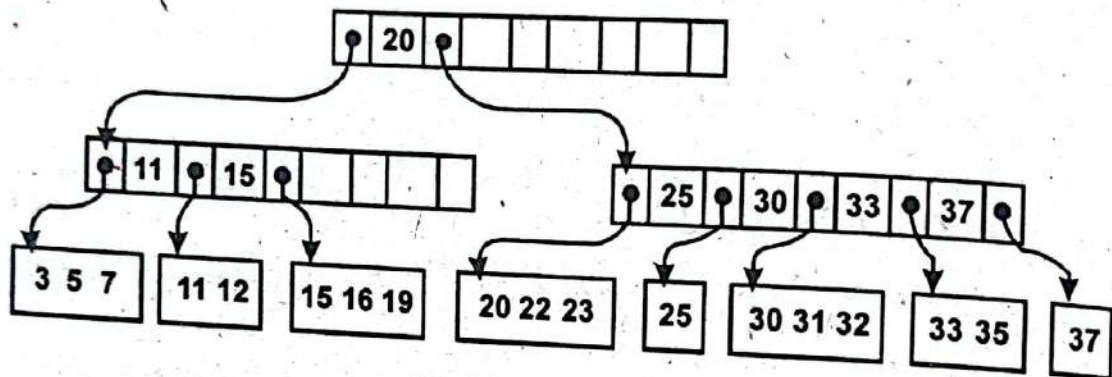


Figure 11.35: B-Tree of order 4

A *B-tree* of order M has the following properties:

- i) The root node is either a leaf node or has children between 2 and M .
- ii) All non-leaf nodes except root (i.e. all internal nodes) have children between $\lceil M/2 \rceil$ and M .
- iii) A non-leaf node with ' n ' children contains ' $n-1$ ' keys.
- iv) All leaf nodes are at the same level (or depth).
- v) All the key values within a node are in ascending order.

When data is inserted or removed from a node, its number of child nodes changes. In order to maintain the pre-defined range, internal or interior nodes may be joined or split. Because a range of child nodes is permitted, *B-trees* do not need re-balancing as frequently as other self-balancing search trees but may waste some space, since nodes are not entirely full. The lower and upper bounds on the number of child nodes are typically fixed for a particular implementation. For example, in a 2-3 tree, each internal node may have only 2 or 3 child nodes.

11.4.1 Operations on B-Tree

The following basic operations are performed on the *B-tree*:

- i) Search
- ii) Insertion
- iii) Deletion

11.4.1.1 Search Operation in B-Tree

In a *B-tree*, the search operation is similar to that of a *binary search tree*. In a binary search tree, we start the search process from the root node, and every time we make a 2-way decision (we go to either left subtree or right subtree). In *B-tree* also we start the search process from the root node but every time we make an *n*-way decision where '*n*' indicates the total number of children of a node. The correct child is chosen by performing a linear search of the values in the node. In a *B-tree*, the search operation is performed with $O(\log n)$ time complexity.

In a *B-tree*, the search operation is performed as follows:

- 1- Input the search element.
- 2- Compare, the search element with the first key value of the root node in the tree.
- 3- If both are matching, then display "Given node found!" and terminate the search process.
- 4- If both are not matching, then check whether the search element is smaller or larger than the key value.
- 5- If the search element is smaller, then continue the search process in the left subtree.
- 6- If the search element is larger, then compare it with the next key value in the same node.
- 7- Repeat steps 3, 4, 5, and 6 until we found an exact match or comparison completed with the last key value in a leaf node.
- 8- If we completed with the last key value in a leaf node and given node not found, then display "Element is not found" and terminate the search process.

11.4.1.2 Insertion Operation in B-Tree

In a *B-tree*, the new element is inserted only at the leaf node. In a *B-tree*, the insertion operation is performed with $O(\log n)$ time complexity. Following steps are performed to insert a new element at the leaf node:

- 1- Check whether the tree is empty.
- 2- If the tree is empty, then create a new node, assign a new key value to it, and set this node as a root node of the tree.
- 3- If the tree is not empty, then search the tree to see if the new element to be inserted is already in the tree.
 - If the element already exists in the tree, then display the message "Element already exists" and terminate the insertion process.
 - If the element is not found in the tree, then the search process terminates at the leaf node.
 - If there is room (empty space) in this leaf node, insert the new element here by maintaining the ascending order of key value within the node.

- If that leaf node is already full (i.e. no room to insert new element), then split it into three parts:
 - i) **Left:** It consists of first $(M-1)/2$ left values
 - ii) **Middle:** It consists of middle value which is found as:

$$1 + (M-1)/2$$
 - iii) **Right:** It consists of last $(M-1)/2$ values

Left and Right become nodes; Middle is added to the node above, with Left and Right as its children. If the node does not have a parent (i.e. it is a root node), then the middle value becomes a new root node for the tree.

Note that the values less than the middle value (or median value) are put in the new left node and the values greater than the middle value are put in the new right node. The middle value will act as a separation value. Repeat the same until the middle value is fixed into a node. It means that if the parent node has no room, then it may have to be split as well.

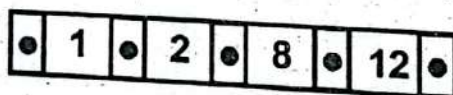
Example:

Construct a B-tree of order 5 by inserting key values in the following order:

1, 12, 8, 2, 25, 6, 14, 28, 17, 7, 52, 16, 48, 68, 3

Perform the following steps to construct the B-tree:

- 1- Insert the first four values or items (such as 1, 12, 8, 2) into the root node by maintaining the ascending order of key values within the node.



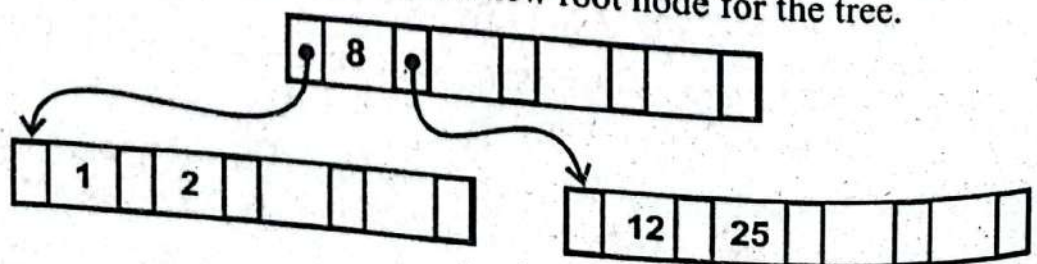
- 2- Now when we try to insert a fifth value or item (such as 25) into the root node, an overflow will occur because there are five values [1, 2, 8, 12, 25]. Four values [1, 2, 8, 12] are already put into the node and there is no room to adjust the fifth value [25] in this node. So we split this node into:

Left = [1, 2]

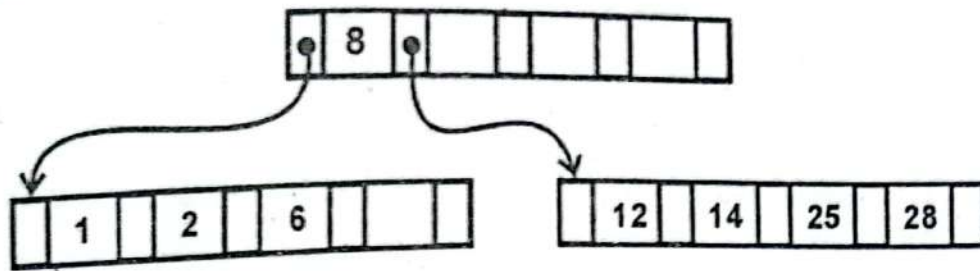
Middle = 8

Right = [12, 25]

Left and Right become nodes; Middle is added to the node above, with Left and Right as its children. But here, this node does not have a parent (i.e. it is a root node). So the middle value becomes a new root node for the tree.



- 3- Insert the next three values or items (such as 6, 14, 28) by maintaining the ascending order of key values within the node. Put value 6 into the left node because this value is less than the value of the parent node and values 14 & 28 into the right node because these values are greater than the value of the parent node.



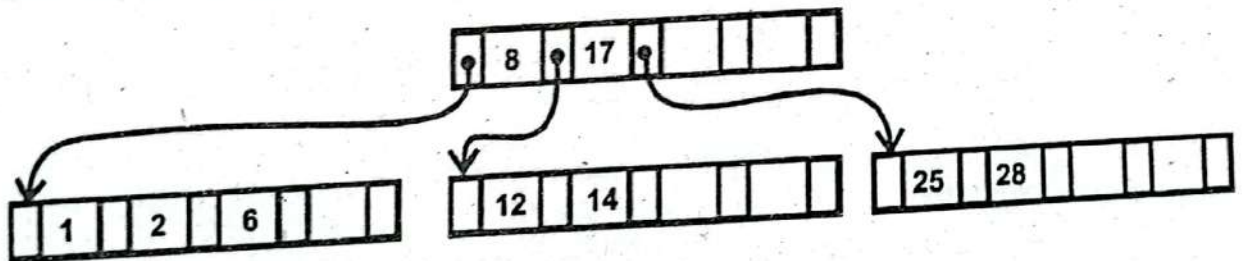
- 4- Next value to be inserted is 17 which is greater than the value of its parent node i.e. '8'. It will be inserted into the right child node. There is no room in the right child node and an overflow will occur because there are five values [12, 14, 17, 25, 28]. So we split this node into:

Left = [12, 14]

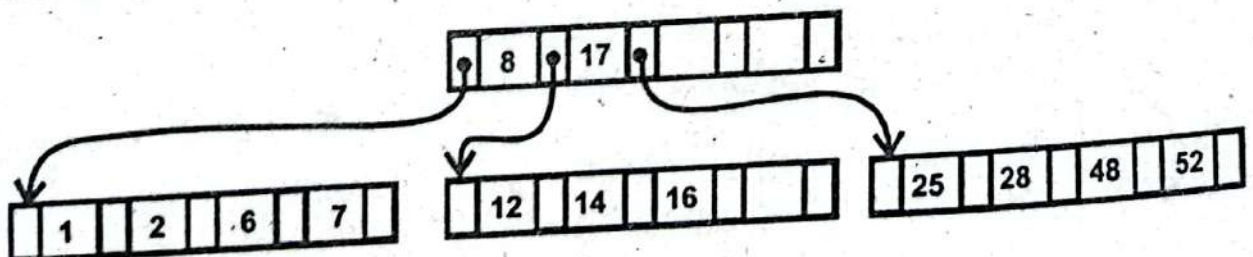
Middle = 17

Right = [25, 28]

Left and Right become nodes; Middle is added to the node above.



- 5- Insert the next four values or items (such as 7, 52, 16, 48) by maintaining the ascending order of key values within the node.



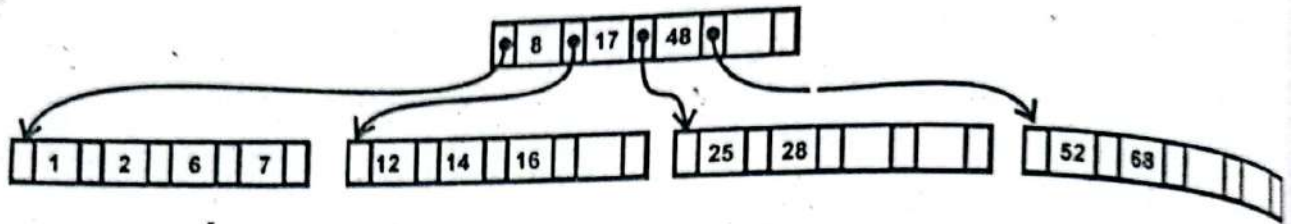
- 6- Next value to be inserted is 68 which is greater than parent value 17, so it will be inserted into its right node. There is no room in the right node and an overflow will occur because there are five values [25, 28, 48, 52, 68]. So we split this node into:

Left = [25, 28]

Middle = 48

Right = [52, 68]

Left and Right become nodes; Middle is added to the node above.



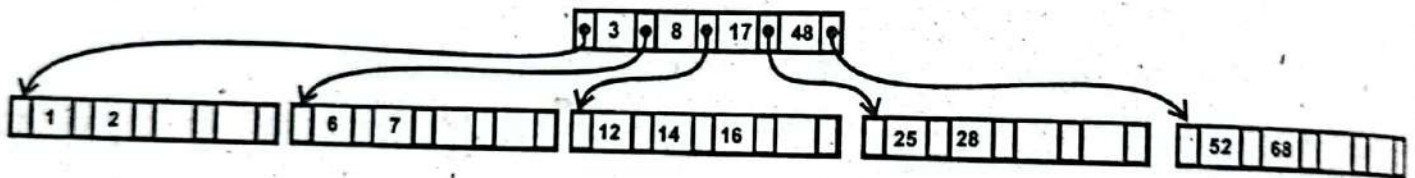
- 7- The last value to be inserted is 3 which is less than parent value 8, so it will be inserted into its left node. There is no room in the right node and an overflow will occur because there are five values [1, 2, 3, 6, 7]. So we split this node into:

Left = [1, 2]

Middle = 3

Right = [6, 7]

Left and Right become nodes; Middle is added to the node above.



11.4.1.3 Deletion Operation in B-Tree

Deletion from a *B-tree* is more complicated than insertion because when we delete a key from an internal node, we will have to rearrange the children of the node. The key value to be deleted may be located in the leaf node or non-leaf node i.e. internal node. Like insertion operation in B-tree, the time complexity of deletion operation is also $O(\log n)$.

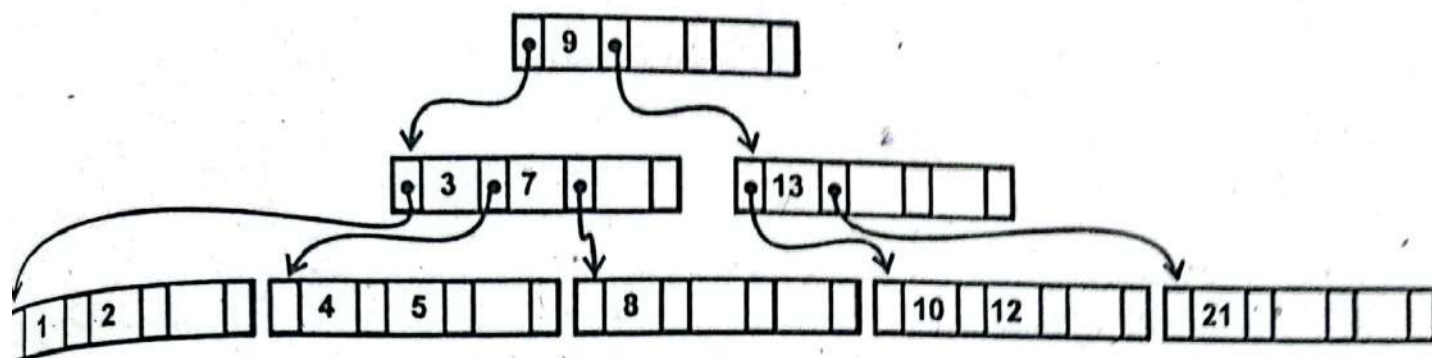
i) Deletion from a Leaf Node

Deleting a value from a leaf node is very simple and easy. Following steps are performed to delete a value from the leaf node:

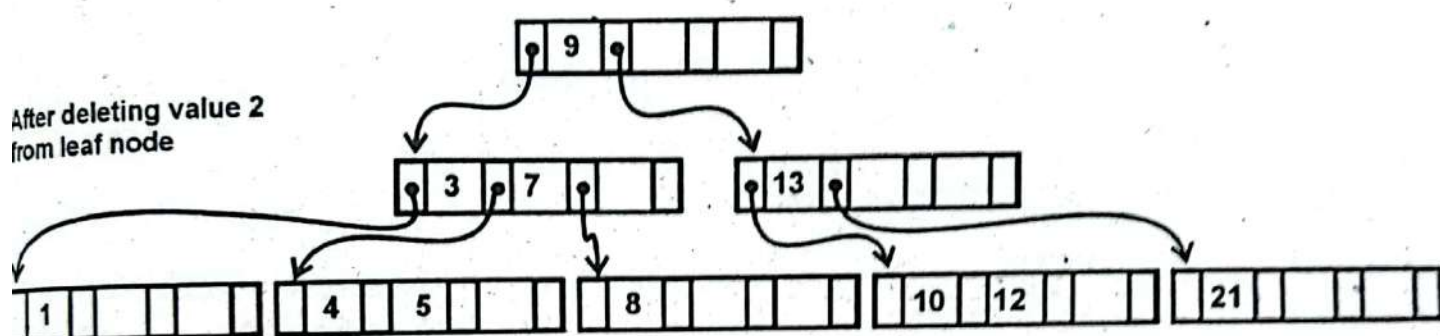
- Search for the value to delete.
- If the value is in a leaf node, it can simply be deleted from the node.
- If an underflow occurs, rebalance the tree. Rebalancing starts from the leaf node and proceeds towards the root until the tree is balanced. If deleting a value from a node has brought it under the minimum size, then some values must be redistributed to bring all nodes up to the minimum. Usually, the redistribution involves moving value from a sibling node that has more than the minimum number of nodes. If no sibling can spare a value, then the deficient node must be merged with a sibling. The merge causes the parent to lose a separator value, so the parent may become deficient and need rebalancing. The merging and rebalancing may continue up to the root node.

Example:

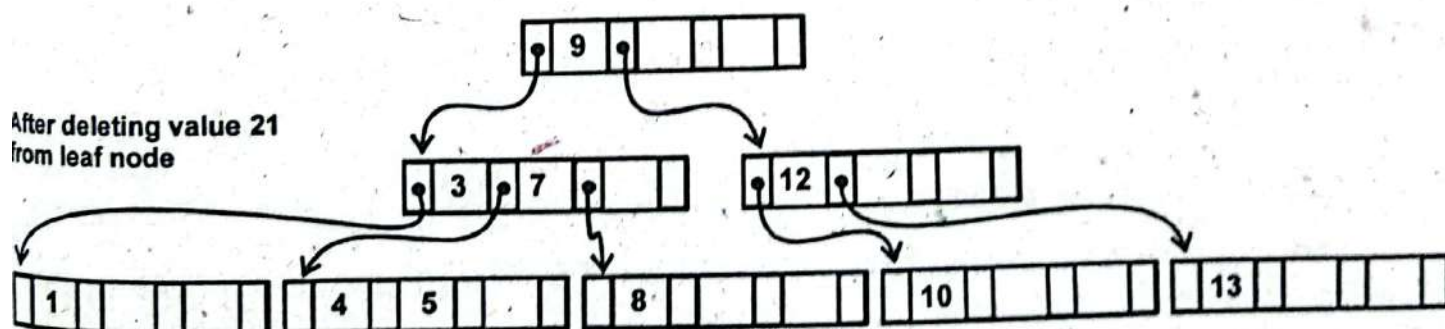
Let us take a B-tree of order 4 as shown in the following figure:



Suppose we delete the value 2 from the leaf node. It can simply be deleted from the node and underflow will occur.



Now suppose we want to delete the value 21 from the leaf node. In this case, underflow will occur. Rebalance the tree by redistributing the values among the nodes.



ii) Deletion from a Non-leaf Node

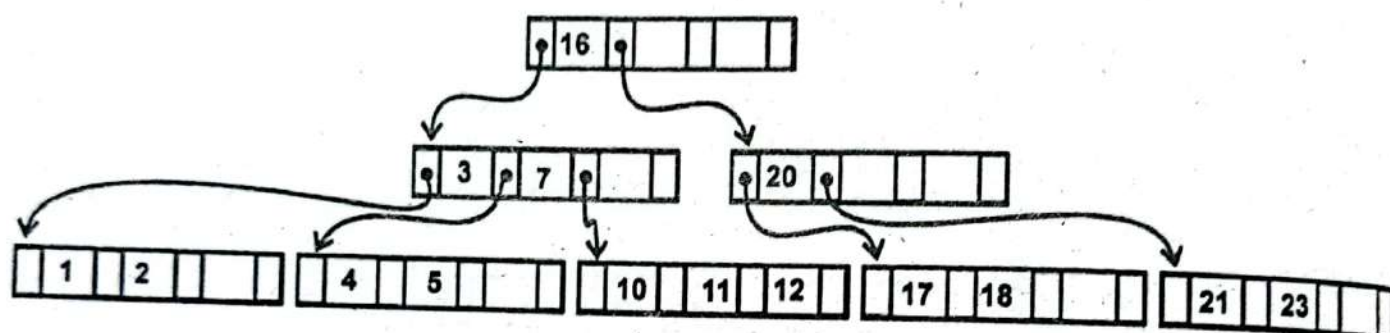
Deleting a value from a non-leaf node is complicated. Each value in an internal node acts as a separation for two subtrees. Therefore, we need to find a replacement for separation. Note that the largest value in the left subtree is still smaller than the separator. Likewise, the smallest value in the right subtree is still greater than the separator. Both of those values are in leaf nodes, and either one can be the new separator for the two subtrees.

Following steps are performed to delete a value from the non-leaf node:

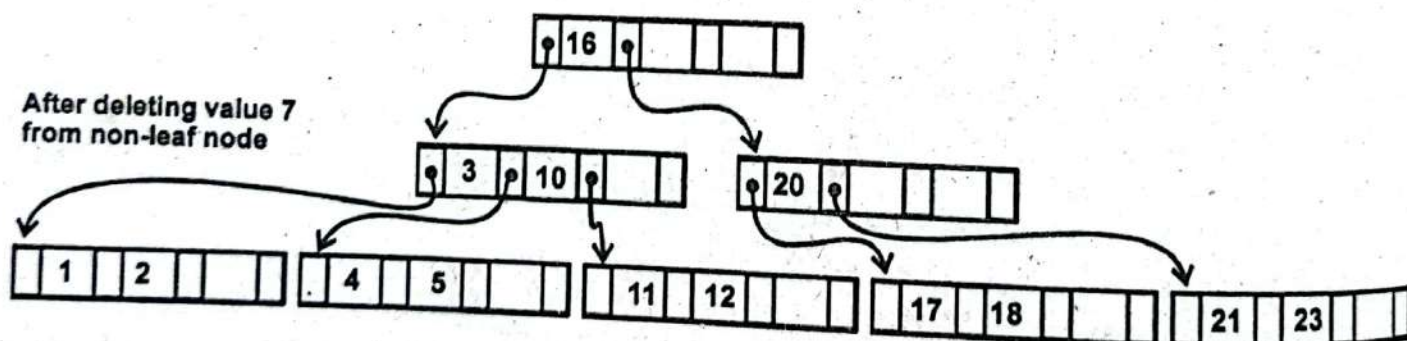
- Choose a new separator value, either the largest value in the left subtree or the smallest value in the right subtree.
- Remove the new selected separator value from the leaf node, and replace this value with the deleted value from parent node. This value becomes the new separator.
- If the leaf node (from where the new value is selected for new separator) is now deficient, then rebalance the tree starting from the leaf node.

Example:

Let us take a B-tree of order 4 as shown in the following figure:



Suppose we want to delete the value 7 from the non-leaf node. First of all, this value will be deleted from the node and then the first value of the right child node will be shifted in that node (its parent node).



Summary --- Complexities of Self-Balancing Trees

AVL Tree

Operation	Time Complexity		Space Complexity
	Average-Case	Worst-Case	
Access	$\Theta(\log n)$	$O(\log n)$	Worst-Case
Search	$\Theta(\log n)$	$O(\log n)$	$O(n)$
Insertion	$\Theta(\log n)$	$O(\log n)$	$O(n)$
Deletion	$\Theta(\log n)$	$O(\log n)$	$O(n)$

Red-Black Tree

Access	$\Theta(\log n)$	$O(\log n)$	$O(n)$
Search	$\Theta(\log n)$	$O(\log n)$	$O(n)$
Insertion	$\Theta(\log n)$	$O(\log n)$	$O(n)$
Deletion	$\Theta(\log n)$	$O(\log n)$	$O(n)$

Splay Tree

Search	$\Theta(\log n)$	$O(\log n)$	$O(n)$
Insertion	$\Theta(\log n)$	$O(\log n)$	$O(n)$
Deletion	$\Theta(\log n)$	$O(\log n)$	$O(n)$

B-Tree

Access	$\Theta(\log n)$	$O(\log n)$	$O(n)$
Search	$\Theta(\log n)$	$O(\log n)$	$O(n)$
Insertion	$\Theta(\log n)$	$O(\log n)$	$O(n)$
Deletion	$\Theta(\log n)$	$O(\log n)$	$O(n)$

Short Answers to the Questions

- ✓ **What is a self-balancing binary search tree?**
 A self-balancing binary search tree is a height-balanced binary search tree. It automatically keeps height as small as possible when insertion and deletion operations are performed on the tree. The height is typically maintained in order of $\log n$ so that all operations take $O(\log n)$ time on average.
- Why AVL tree is better than binary search tree?**
 AVL tree is also a BST but it can rebalance itself. This behavior makes it faster in the worst-case. It keeps rebalancing itself so in worst-case it takes $O(\log n)$ time, while the BST takes $O(n)$. So, it is always better to implement an AVL tree than BST.
- What is the difference between the AVL tree and the Red-Black tree?**
 Following are the main differences between the AVL tree and the Red-Black tree
- 1- AVL tree provides faster lookups than the Red-Black tree because it is more strictly balanced.
 - 2- Red-Black tree provides faster insertion and removal operations than the AVL tree as fewer rotations are performed due to relatively relaxed balancing.
 - 3- AVL tree stores balance factors or heights for each node, thus requires $O(n)$ extra space, whereas Red-Black tree requires only 1 bit of information per node, thus require $O(1)$ extra space.

- 4- Red-Black tree is used in most of the language libraries like a map, multimap, multiset in C++, whereas AVL trees are used in databases where faster retrievals are required.

 What is the difference between B-tree and binary search tree?

In B-tree, a non-terminal node can have the maximum M number of child nodes, where M is the order of the B-tree. On the other hand, a Binary tree can have at most two child nodes or subtrees. B-tree is used when data is stored in disk, whereas a binary tree is used when data is stored in fast memory like RAM.

 What is the difference between the splay tree and AVL tree?

Both splay tree and AVL tree are binary search trees with excellent performance guarantees, but they differ in how they achieve those guarantees. In an AVL tree, the shape of the tree is constrained at all times such that the tree shape is balanced. This shape is maintained on insertions and deletions and does not change during lookups. Splay tree, on the other hand, maintains efficient by reshaping the tree in response to lookups on it. That way, frequently-accessed elements move up toward the top of the tree and have better lookup times. The shape of splay trees is not constrained and varies based on what lookups are performed.

 What is the application of splay trees?

Splay trees are typically used in the implementation of caches, memory allocators, garbage collectors, data compression, ropes (replacement of string used for long text strings), in Windows NT (in the virtual memory, networking, and file system code), etc.

EXERCISE

Q#1 Select the correct answer from the multiple choices.

- What is an AVL tree?
 - a tree which is balanced and is a height balanced tree
 - a tree which is unbalanced and is a height balanced tree
 - a tree with three children
 - a tree with at most 3 children
- The term "balance factor" is used in:
 - Red-Black Tree
 - Splay Tree
 - AVL Tree
 - B-Tree
- What is the maximum height of an AVL tree with 'n' nodes?
 - n
 - $\log(n)$
 - $\log(n/2)$
 - $n/2$
- What maximum difference in heights between the leafs of a AVL tree is possible?
 - $\log(n)$, where n is the number of nodes n
 - n, where n is the number of nodes
 - 0 or 1
 - at most 1

Why to prefer red-black trees over AVL trees?

- (a) Because red-black is more rigidly balanced
- (b) AVL tree store balance factor in every node which costs space
- (c) AVL tree fails at scale
- (d) Red black is more efficient

Which of the following is a self-adjusting or self-balancing Binary Search Tree?

- (a) Splay Tree
- (b) AVL Tree
- (c) Red Black Tree
- (d) All

Balance factor of every node of AVL tree is either

- (a) 1, -1, -2
- (b) 0, 1, -1
- (c) 0, 1, 2
- (d) 0 or -1, -2

The balance factor of left-heavy AVL tree is:

- (a) -1
- (b) 0
- (c) 1
- (d) 2

Which one is single rotation?

- (a) LL rotation
- (b) LR rotation
- (c) RL rotation
- (d) All

Which of the following is double rotation?

- (a) LR rotation
- (b) RR rotation
- (c) RL rotation
- (d) Both a & c

1. The Worst-case time complexity of insertion operation of AVL tree is:

- (a) $O(n)$
- (b) $O(n \log n)$
- (c) $O(\log n)$
- (d) $O(n^2)$

2. In Red-Black tree, if a node is red, then:

- (a) both its children are red.
- (b) both its children are black.
- (c) left node black and right node red
- (d) left node red and right node black

3. The Worst-case space complexity of Red-Black tree is:

- (a) $O(n)$
- (b) $O(n \log n)$
- (c) $O(\log n)$
- (d) $O(n^2)$

4. In splay tree, the Zig step is equivalent to _____ rotation of AVL tree.

- (a) RL
- (b) LR
- (c) RR
- (d) LL

5. In splay tree, the Zag step is equivalent to _____ rotation of AVL tree.

- (a) RL
- (b) LR
- (c) RR
- (d) LL

6. Which of the following is not a binary tree?

- (a) Splay Tree
- (b) Red-Black Tree
- (c) B-Tree
- (d) AVL Tree

7. In a B-tree of order 4, each non-leaf node contains:

- (a) maximum 3 keys and maximum 4 children.
- (b) maximum 2 keys and maximum 3 children.
- (c) maximum 4 keys and maximum 3 children.
- (d) maximum 4 keys and maximum 4 children.

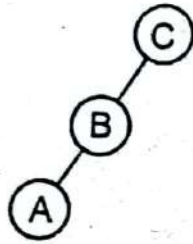
8. A non-leaf node of B-tree with '4' children contains:

- (a) 4 keys
- (b) 3 keys
- (c) 3 to 4 keys
- (d) None

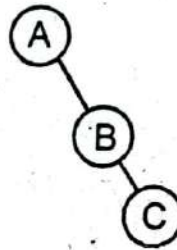
Answers of MCQs

01 (a)	02 (c)	03 (b)	04 (a)	05 (b)	06 (d)	07 (b)	08 (c)	09 (a)	10 (d)
11 (c)	12 (b)	13 (a)	14 (c)	15 (d)	16 (c)	17 (a)	18 (b)		

- Q.2 What is an AVL tree? Explain with a suitable example.
- Q.3 Why rotation operation is performed on AVL tree? Discuss different types of rotations.
- Q.4 Following trees are unbalanced. How these trees can be balanced? Explain.



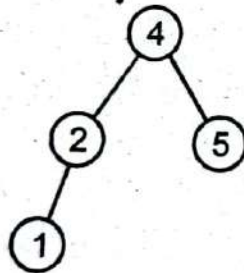
Not balanced



Not balanced

- Q.5 Define a splay tree. Explain splaying with an example.

- Q.6 Consider the following binary tree:



Suppose we insert a new node with value 3. Explain the splaying process of new node 3 with diagrams.

- Q.7 ★ Explain B-tree with example.

As we have seen in *tree* data structures, one parent node may have no child, one child, or many child nodes. Therefore, the *tree* data structure represents one-to-many relationships. In real life, however, we frequently come across problems that can be best described by many-to-many relationships. Such problems cannot be solved using trees or other data structures we have studied so far. To deal with such problems, the graph data structure is used. Note that the tree is also a graph.

12.1 GRAPHS

Graph¹ is a non-linear data structure. It consists of a finite set of points or nodes and a set of links that connects the points. The points are called *vertices* and the links that connect the vertices are called *edges*. In a graph, the vertices are represented by small circles, while edges are represented by lines or arcs. The vertices of a graph can be used to represent objects and the edges represent the relationships between objects. For example, the vertices might represent cities and the edges represent roads that link different cities.

A set of vertices $\{v_1, v_2, \dots, v_n\}$ is denoted by a letter V , while a set of edges $\{e_1, e_2, \dots, e_n\}$ is denoted by a letter E . An edge is a pair of vertices, which is represented as:

$$e = (v_i, v_j)$$

Where ' v_i ' and ' v_j ' denote the start vertex and end vertex of edge ' e ', respectively. They are known as the *head point* and *tail point* of the edge ' e '. They are also known as *end vertices* or *endpoints* of the edge ' e '.

Generally, a graph G is a pair of sets (V, E) , where V is the set of vertices and E is the set of edges, connecting the pairs of vertices.

Consider the following graph as shown in figure 12.1:

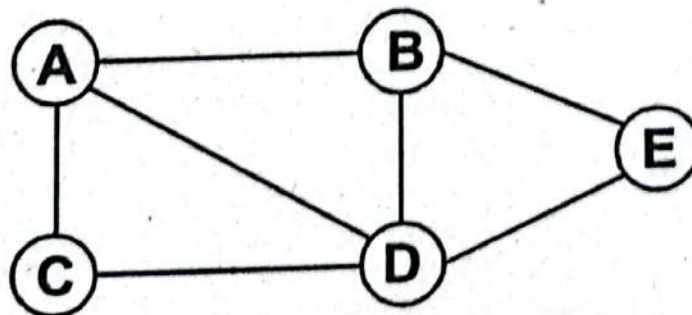


Figure 12.1: Graph showing Vertices and Edges

¹ A graph is a pictorial representation of a set of objects where some pairs are connected by links.

In figure 12.1, the labeled circles represent vertices. Thus, A, B, C, D, and E are vertices, whereas the lines from A to B, A to D, A to C, and so on represent edges. This graph G consists of 5 vertices and 7 edges. It can be defined as:

$$G = (V, E)$$

Where

$$V = \{A, B, C, D, E\}$$

$$E = \{(A, B), (A, C), (A, D), (B, D), (C, D), (B, E), (E, D)\}$$

12.1.1 Basic Terminologies Related to Graphs

The basic terminologies related to graphs are as follows:

1- Vertex

An individual data element (or node) of a graph is called the *vertex*. In a graph, as shown in figure 12.1, the labeled small circles represent vertices. Thus, A, B, C, D, and E are vertices of this graph.

2- Edge

A line that connects vertices in a graph is called an *edge*. For example, in the graph, as shown in figure 12.1, the line from A to B represents an *edge*. There are three types of edges.

- **Directed Edge:** A type of edge that has an arrow sign at its one end is called a *directed edge*. It means that a directed edge shows a direction from one vertex to another. It is a unidirectional edge. If there is a directed edge between vertices A and B, then edge (A, B) is not equal to the edge (B, A).
- **Undirected Edge:** A type of edge that has no arrow is called an *undirected edge*. It means that the undirected edge shows no direction from one vertex to another. It is a bidirectional edge. If there is an undirected edge between vertices A and B, then edge (A, B) is equal to the edge (B, A).
- **Weighted Edge:** A type of edge that has a weight, number, or any value associated with it is called a *weighted edge*. The weight of an edge is sometimes also called its *cost*. The weight of an edge usually represents certain conditions or situations associated with the edge. For example, in a road network, the weight of each road may denote distances between two cities.

3- Directed Graph

A graph that has only directed edges from one vertex to another is called a *directed graph*. A directed graph is also called a *digraph*. Figure 12.2 shows a directed graph. This graph consists of four vertices numbered 1, 2, 3, 4, and four directed edges joining vertices 1 to 2, 2 to 4, 4 to 3, and 3 to 1.

The directed graphs are commonly used to analyze electrical circuits, develop project schedules, find shortest routes, analyze social relationships, and construct models for analysis and solution of many other problems.

4✓ Undirected Graph

A graph that has only undirected edges is called an *undirected graph*. An undirected graph is also called an *undigraph*. Figure 12.3 shows an undirected graph.

Some graphs have both directed and undirected edges. These graphs are called *mixed graphs*.

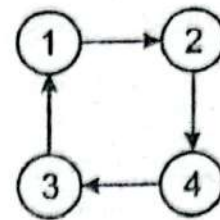


Figure 12.2: Directed graph

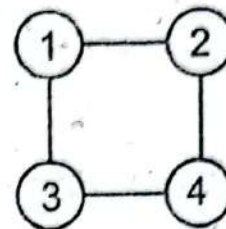


Figure 12.3: Undirected graph

5✓ Weighted Graph

A graph that has only weighted edges is called a *weighted graph*. Figure 12.4 shows a weighted graph. An edge of a weighted graph is represented as:

$$e = [v_i, v_j, w]$$

Where ' v_i ' and ' v_j ' denote the start vertex and end vertex of edge ' e ' respectively, whereas ' w ' represents the weight of the corresponding edge.

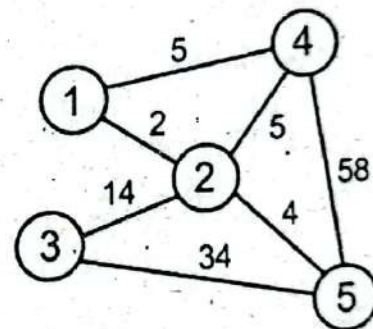


Figure 12.4: Weighted graph

6✓ Degree of a Vertex

The number of edges connected to a vertex is called the *degree* of that vertex. In the graph shown in figure 12.5, vertices A, B & D have a degree of 3, vertex E has a degree of 2, and vertex C has a degree of 1.

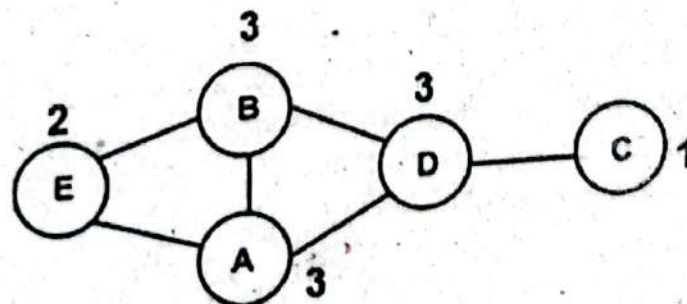


Figure 12.5: Graph showing Degree of Vertex

A vertex that has degree 0 is called an *isolated vertex*. A graph that has only one isolated vertex is called a *null graph*.

7- In-Degree of a Vertex

In a directed graph, the number of incoming edges connected to a vertex is called *in-degree* of that vertex. A directed edge is said to be an incoming edge on its destination vertex. In the directed graph shown in figure 12.6, the in-degree of vertex A is 0, and vertex B has an in-degree of 2. Similarly, in-degrees of other vertices are shown.

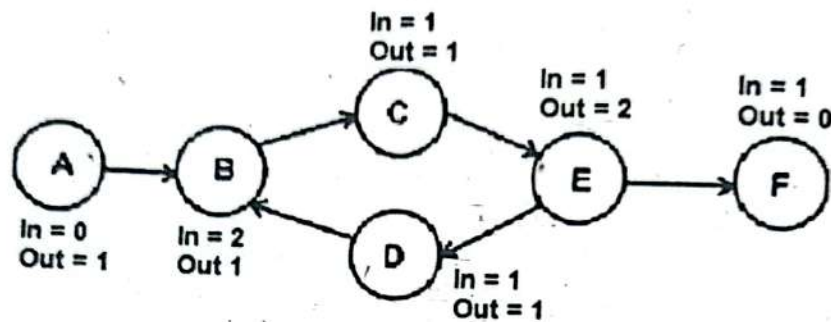


Figure 12.6: Graph showing In-Degree and Out-Degree

8- Out-Degree of a Vertex

The number of outgoing edges connected to a vertex is called the *out-degree* of that vertex. A directed edge is said to be an outgoing edge on its source vertex. In the directed graph shown in figure 12.6, the out-degree of vertices A, B, C & D is 1. Similarly, vertex E has an out-degree of 2. The vertex that has an out-degree of 0 is called *terminal vertex* or *leaf* and other vertices are called the *branch vertices*. In figure 12.6, F is the terminal vertex.

The sum of the out-degree and in-degree of a vertex is the *total degree* of that vertex. The total degree of a loop vertex is 2 and that of an isolated vertex is 0.

9- Source & Sink Vertices

The vertex that has a positive out-degree but zero in-degree is called the *source vertex*. In the graph shown in figure 12.7, vertex B is the source vertex. This vertex has a positive out-degree of 3 and its in-degree is zero.

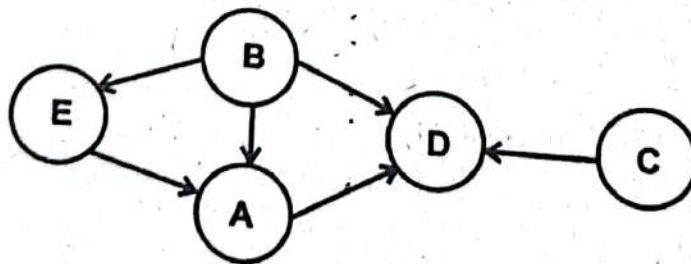


Figure 12.7: Graph showing Source & Sink Vertices

The vertex that has zero out-degree but a positive in-degree is called *sink vertex*. In the graph shown in figure 12.7, vertex D is the sink vertex. This vertex has a positive in-degree of 3 and its out-degree is zero.

10✓ Path & Length of a Graph

A path in a graph is a sequence of edges between vertices. In a graph shown in figure 12.8, ABCD represents the path from vertex A to vertex D. Similarly, AFG represents the path from vertex A to vertex G.

The number of edges in a path of a graph is called the *length of the graph*. For a path consisting of 'n' vertices, the path length will be $n-1$.

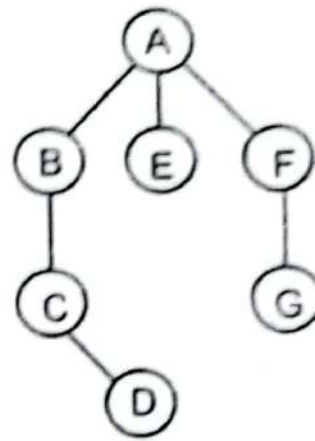


Figure 12.8: Graph showing Path

11✓ Loop Edge

An edge 'e' is said to be a loop edge if it has the same vertex at its tail and head. A loop edge is shown in figure 12.9.

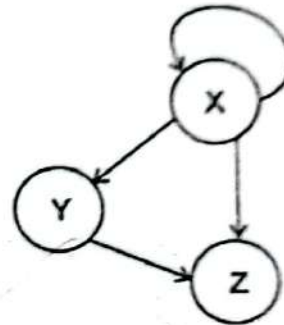


Figure 12.9: Loop Edge

12✓ Multiple or Parallel Edges

The edges that have the same tail and head vertices are known as multiple or *parallel edges*. Multiple edges are shown in figure 12.10.

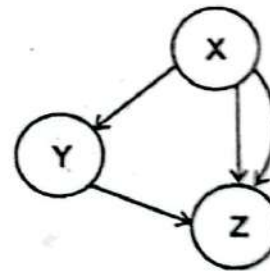


Figure 12.10: Multiple Edges

13✓ Simple Graph

A graph is said to be a simple graph if it does not contain any loop-edge (or self-loop).

14✓ Cycle & Acycle

A path that originates or begins and ends at the same vertex is called a cycle. In other words, a path from a particular vertex up to itself is called a cycle. It is also known as a *circuit*. The length of a cycle must be at least 1. In figure 12.12, an example of a cycle is given.

The directed graph that has no cycles is called an acyclic graph. A directed acyclic graph is also known as *DAG*.

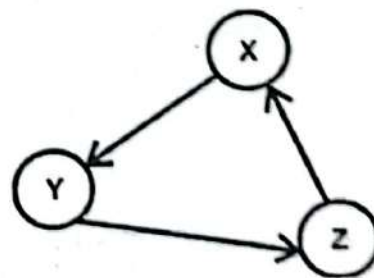


Figure 12.12: Graph showing Cycle

15✓ Adjacent Vertices

Two vertices are adjacent if they are connected to each other by an edge. In a graph shown in figure 12.8, vertices A and B are adjacent but vertices B and E are not adjacent.

16✓ Adjacent Edges

Two edges are adjacent if they share the same vertex. In figure 12.13, edges AB and BC are adjacent because they share the common vertex B.

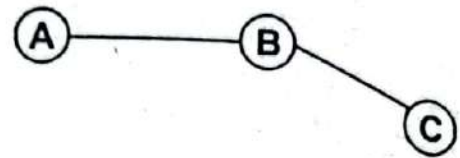


Figure 12.13: Adjacent Edges

17✓ Complete Graph

A graph is said to be a complete graph if there is an edge between every pair of vertices, as shown in figure 12.14. In other words, a graph in which all pairs of vertices are adjacent is called a *complete graph*. This type of graph is also called a *mesh graph*.

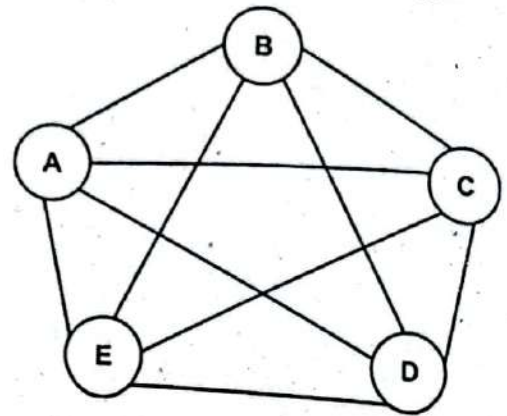


Figure 12.14: Complete Graph

18✓ Sparse Graph

A graph that has relatively few edges when compared to the number of vertices is called a *sparse graph*. Examples of sparse graphs are transportation and road networks where the intersections are vertices and roads are edges. For such networks, the number of roads is not significantly larger than the number of intersections.

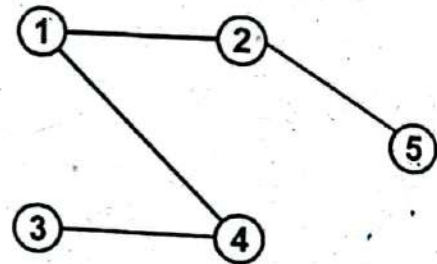


Figure 12.15: Sparse Graph

19✓ Dense Graph

A graph that has relatively many edges when compared to the number of vertices is called a *dense graph*. A complete graph can be a dense graph but the reverse may not be true.

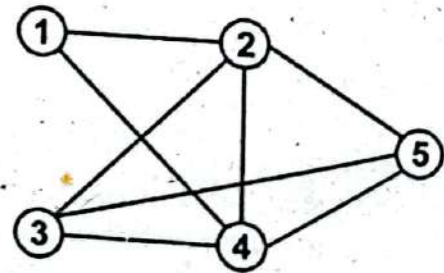


Figure 12.16: Dense Graph

Reading

12.1.2 Applications of Graphs

Graphs are used to study and modeling of real-world systems in different areas like computer sciences, electrical engineering, chemistry, physics, biology, computer chip design, and many other fields. Following are some examples where graphs are commonly used:

- ★ **Computer Networks:** Graphs are used for modeling computer networks of communication and computational devices. For instance, the link structure of a website can be represented by a directed graph, in which the vertices represent web pages and directed edges represent links from one page to another. Similarly, in networks' routing, servers represent vertices, data links represent edges, and connection speed represents weight.

Graphs are also used in social networks like Facebook and LinkedIn. For example, on Facebook, each person is represented with a vertex (or node). Each vertex is a structure and contains information like a person's id, name, gender, and locale, etc.

- ★ **Finding Path:** Graphs are used for finding a path from a starting condition to a goal condition; for example, problem-solving in artificial intelligence. Graphs are also used in GPS navigation in which road intersections represent vertices, roads represent edges, and time required to go from one intersection to another represents weight.
- ★ **Computer Algorithms:** Graphs are used for modeling computer algorithms, showing transitions from one program state to another.
- ★ **Family Tree:** Graphs are also used for modeling family trees.
- ★ **Modeling City Map:** A map of a city can be modeled by a weighted graph. On each street, the edge is compared with a length, corresponding to the length of the street, and direction – the direction of movement. If a street is two-way, it can be compared to two edges in both directions. At each intersection, there is a vertex. In such a model there are natural tasks such as searching for the shortest path between two intersections, checking whether there is a road between two intersections, checking for a loop (if we can turn and go back to the starting position) searching for a path with a minimum number of turns, etc.
- ★ **Modeling River System:** River system in a given region can be modeled by a weighted directed graph, where each river is composed of one or more edges, and each vertex represents the place where two or more rivers flow into another one. On the edges, values can be set, showing the amount of water that flows through them. Naturally, in this model, there are tasks such as calculating the volume of water, passing through each vertex, and anticipation of the possible flood where the volume of water is increasing.
- ★ **Transportation Network:** In transportation networks, graph vertices correspond to locations where people or goods are moved around. The locations may represent cities, warehouses, airports, and terminals. The edges correspond to the connections between the vertices. The edges may represent roads, railway tracks, air routes, etc. along which transportation in cities takes place. Figure 12.17 represents transportation links between ten cities. The vertices represent cities and the edges correspond to the roads that link the cities. The graph shown in this figure has 10 vertices and 15 edges.
- ★ **Chemistry and Physics:** Graph theory is also used to study molecules in chemistry and physics. In condensed matter physics, the three-dimensional structure of complicated simulated atomic structures can be studied quantitatively by gathering statistics on graph-theoretic properties related to the topology of the atoms. In chemistry, a graph can show a natural model of a molecule, where vertices represent atoms, and edges represent bonds. This approach is especially used in computer processing of molecular structures, ranging from chemical editors to database searching. In statistical physics, graphs can represent local connections between interacting parts of a system, as well as the dynamics of a physical process on such systems. Similarly, in computational neurosciences, graphs can be used to represent

functional connections between brain areas that interact to give rise to various cognitive processes, where the vertices represent different areas of the brain and the edges represent the connections between those areas.

- ★ **Biology:** Graph theory is useful in biology and conservation efforts where a vertex can represent regions where certain species exist (or inhabit) and the edges represent migration paths or movement between different regions. This information is important when looking at breeding patterns or tracking the spread of disease, parasites, or how changes to the movement can affect other species.
- ★ **Mathematics:** In mathematics, graphs are useful in geometry and certain parts of topology such as knot theory. Algebraic graph theory has close links with group theory.

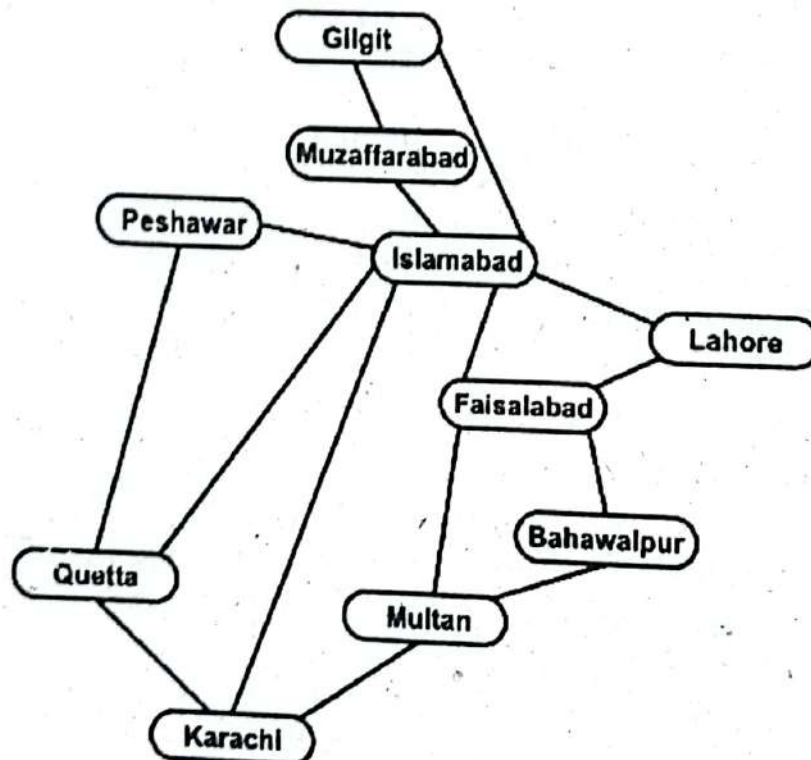


Figure 12.17: Transportation links between important cities of Pakistan

12.2 REPRESENTATION OF GRAPHS

Graphs are unstructured. One vertex in a graph might be adjacent to many other vertices. Similarly, a vertex might only be adjacent to just one vertex. This property of adjacency is used to represent graphs in computer memory. There are two common ways of representing graphs. One way uses the adjacency matrix and the other uses the adjacency list.

12.2.1 Adjacency Matrix Representation

The adjacency matrix is one of the common ways to represent a graph. It is a two-dimensional array of size $V \times V$, where V is the number of vertices in the graph. It means that a graph of 5 vertices can be represented using a matrix (2D) of size 5×5 . The adjacency matrix shows which vertices are adjacent to one another.

In the adjacency matrix, rows and columns both represent vertices. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph. For directed and undirected graphs, each cell of the matrix contains value 1, if a pair of vertices is adjacent; otherwise, the entry is 0.

12.2.1.1 Representation of Undirected Graph

Consider an undirected graph that contains five vertices 1, 2, 3, 4, and 5. The vertex 1 is linked to vertex 4. The vertex 2 is linked to vertices 4 and 5. The vertex 3 is linked to vertex 5. The vertex 4 is linked to vertices 1, 2, and 5. The vertex 5 is linked to vertices 2, 3, and 4.

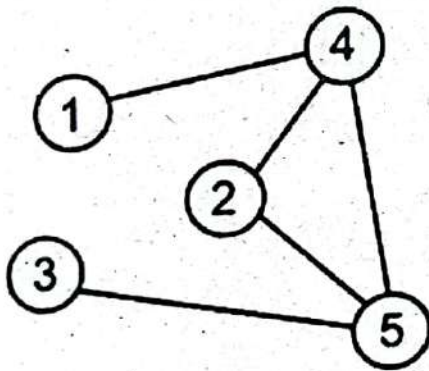
1 - [4]

2 - [4, 5]

3 - [5]

4 - [1, 2, 5]

5 - [2, 3, 4]



	1	2	3	4	5
1	0	0	0	1	0
2	0	0	0	1	1
3	0	0	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0

Adjacency Matrix

Program Code 12.1

// This program implements the above mentioned adjacency matrix representation
// of undirected graph of 5 vertices

```
#include <iostream.h>
#include <conio.h>
```

```
class Graph
{
```

```
private:
    int adjMatrix[6][6];
public:
```

```
    // This function fills value 1 in adjacent pair of vertices
    void addEdge(int i, int j)
    {
```



```

        if ((i>=1 && i<6) && (j>=1 && j<6))
        {
            adjMatrix[i][j] = 1;
            adjMatrix[j][i] = 1;
        }
        else
            cout<<"Invalid pair of edges";
    }
}

```

// This function initializes value 0 into all elements of the matrix

```

void initialize_matrix(void)
{
    for (int r = 0; r<6; r++)
        for (int c = 0; c<6; c++)
            adjMatrix[r][c] = 0;
}

```

// This function prints the values of all elements of the matrix

```

void print_matrix(void)
{
    clrscr();
    for (int r = 1; r<6; r++)
    {
        for (int c = 1; c<6; c++)
            cout<<adjMatrix[r][c]<<" ";
        cout<<endl;
    }
}

```

```
};
```

```
void main()
{
```

```

    Graph g;
    g.initialize_matrix();
    g.addEdge(1, 4);
    g.addEdge(2, 4);
    g.addEdge(2, 5);
    g.addEdge(3, 5);
    g.addEdge(4, 1);
    g.addEdge(4, 2);
    g.addEdge(4, 5);
    g.addEdge(5, 2);
    g.addEdge(5, 3);
    g.addEdge(5, 4);
    g.print_matrix();
    getch();
}

```

Output of the program:

```

0 0 0 1 0
0 0 0 1 1
0 0 0 0 1
1 1 0 0 1
0 1 1 1 0

```

12.2.1.2 Representation of Directed Graph

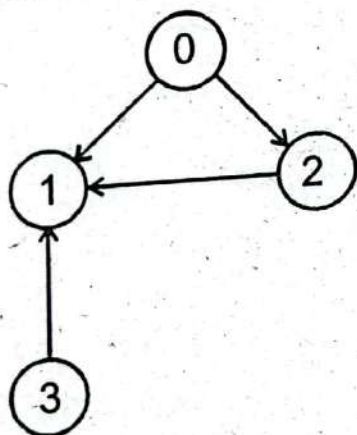
Consider a directed graph that contains four vertices 0, 1, 2, and 3. The vertex 0 is linked to vertices 1 and 2. The vertex 1 is not directly linked to any other vertex. The vertex 2 and vertex 3 are linked to vertex 1.

0 - [1, 2]

1 - [NULL]

2 - [1]

3 - [1]



	0	1	2	3
0	0	1	1	0
1	0	0	0	0
2	0	1	0	0
3	0	1	0	0

Adjacency Matrix

Program Code 12.2

// This program implements the above mentioned adjacency matrix representation
 // of directed graph of 4 vertices

```

#include <iostream.h>
#include <conio.h>

```

```

class Graph
{

```

```

private:
    int adjMatrix[4][4];

```

```

public:

```

```

    // This function fills value 1 in adjacent pair of vertices
    void addEdge(int i, int j)
    {

```

```

        if ((i >= 0 && i < 4) && (j >= 0 && j < 4))
            adjMatrix[i][j] = 1;

```

```

    else

```



```

        cout<<"Invalid pair of edges";
    }

    // This function initializes value 0 into all elements of the matrix
    void initialize_matrix(void)
    {
        for (int r = 0; r<4; r++)
            for (int c = 0; c<4; c++)
                adjMatrix[r][c]= 0;
    }

    // This function prints the values of all elements of the matrix
    void print_matrix(void)
    {
        clrscr();
        for (int r = 0; r<4; r++)
        {
            for (int c = 0; c<4; c++)
                cout<<adjMatrix[r][c]<<" ";
            cout<<endl;
        }
    }

};

void main()
{
    Graph g;
    g.initialize_matrix();
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(2, 1);
    g.addEdge(3, 1);
    g.print_matrix();
    getch();
}

```

Output of the program:

```

0 1 1 0
0 0 0 0
0 1 0 0
0 1 0 0

```

12.2.1.3 Representation of Weighted Graph

For a weighted graph, the adjacency matrix will be populated using the weight as an entry. Consider a weighted graph that contains five vertices 0, 1, 2, 3, and 4. The vertex 0 is linked to vertices 1 and 2. The vertex 1 is linked to vertices 0, 2, and 3. The vertex 2 is linked to vertices 0, 1, and 4. The vertex 3 is linked to vertices 1 and 4. The vertex 4 is linked to vertices 2 and 3.

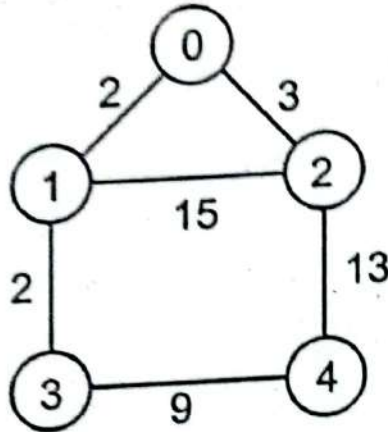
0 - [1, 2]

1 - [0, 2, 3]

2 - [0, 1, 4]

3 - [1, 4]

4 - [2, 3]



	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

Adjacency Matrix

Program Code 12.3

// This program implements the above mentioned adjacency matrix representation
 // of weighted undirected graph of 5 vertices

```
#include <iostream.h>
#include <conio.h>
```

```
class Graph
{
```

```
private:
```

```
int adjMatrix[5][5];
```

```
public:
```

```
// This function fills value 1 in adjacent pair of vertices
```

```
void addEdge(int i, int j, int w)
```

```
{
    if ((i >= 0 && i < 5) && (j >= 0 && j < 5))
```

```
{
        adjMatrix[i][j] = w;
        adjMatrix[j][i] = w;
```

```
}
```

```
else
    cout << "Invalid pair of edges";
```

```
}

// This function initializes value 0 into all elements of the matrix
```

```
void initialize_matrix(void)
```

```
{
    for (int r = 0; r < 5; r++)
        for (int c = 0; c < 5; c++)
            adjMatrix[r][c] = 0;
```



```

    }

    // This function prints the values of all elements of the matrix
    void print_matrix(void)
    {
        clrscr();
        for (int r = 0; r < 5; r++)
        {
            for (int c = 0; c < 5; c++)
                cout << adjMatrix[r][c] << "\t";
            cout << endl;
        }
    }

};

void main()
{
    Graph g;
    g.initialize_matrix();
    g.addEdge(0, 1, 2);
    g.addEdge(0, 2, 3);
    g.addEdge(1, 0, 2);
    g.addEdge(1, 2, 15);
    g.addEdge(1, 3, 2);
    g.addEdge(2, 0, 3);
    g.addEdge(2, 1, 15);
    g.addEdge(2, 4, 13);
    g.addEdge(3, 1, 2);
    g.addEdge(3, 4, 9);
    g.addEdge(4, 2, 13);
    g.addEdge(4, 3, 9);
    g.print_matrix();
    getch();
}

```

Output of the program:

0	2	3	0	0
2	0	15	2	0
3	15	0	0	13
0	2	0	0	9
0	0	13	9	0

12.2.1.4 Advantages and Disadvantages of Adjacency Matrix Representation

Following are the main advantages of adjacency matrix representation:

- It is very simple to implement.
- Adding or removing the time of an edge can be done in $O(1)$ time (constant time). Similarly, the same time is required to check, if there is an edge between two vertices.
- It is very convenient and simple to program.

Following are some disadvantages of adjacency matrix representation:

- It consumes a huge amount of memory for storing big graphs. It takes $O(n^2)$ in memory.
- It requires huge efforts for adding or removing a vertex.
- In many algorithms, we need to know the edges, adjacent to the current vertex. We have to get such information from the adjacency matrix by reading the corresponding row, which results in $O(|V|)$ complexity.

12.2.2 Adjacency List Representation

Adjacency List is one of the most common ways to represent graphs. In the adjacency list, an array of linked list is used. The size of the array is equal to the number of vertices of the graph. Each element of the array represents a specific vertex and a linked list of vertices that are adjacent to that vertex. This kind of graph representation is one of the alternatives to the adjacency matrix. It requires less amount of memory than the adjacency matrix.

Let us take an undirected graph that contains five vertices 0, 1, 2, 3, and 4. The vertex 0 is linked to vertices 1, 2, 3, and 4. The vertex 1 is linked to vertices 0 and 3. The vertex 2 is linked to vertices 0, 3, and 4. The vertex 3 is linked to vertices 0, 1, 2, and 4. The vertex 4 is linked to vertices 0, 2, and 3.

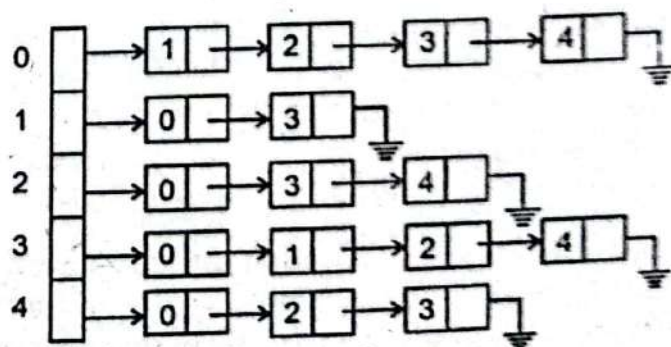
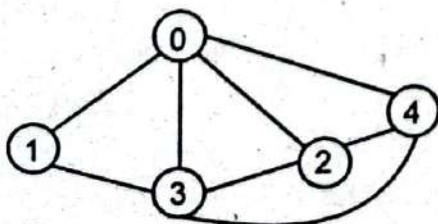
0 - [1, 2, 3, 4]

1 - [0, 3]

2 - [0, 3, 4]

3 - [0, 1, 2, 4]

4 - [0, 2, 3]



Following is the structure of the vertex or node of the graph:


```
struct node
{
    int data;
    struct node* link;
};
```

Following statement declares an array 'V' of type node of size 5:

```
node* V[5];
```

12.2.2.1 Advantages and Disadvantages of Adjacency List Representation

Following are the main advantages of adjacency list representation:

- ★ It does not need to allocate list size and any number of vertices can be created dynamically whenever these are required. Therefore, adjacency list representation saves memory space and easy to maintain.
- ★ It allows to store graph in a more compact form than the adjacency matrix.
- ★ It allows to get the list of adjacent vertices in $O(1)$ time.

Following are some disadvantages of adjacency list representation:

- ❖ Adding or removing edge to and from the adjacency list is not so easy as for adjacency matrix.
- ❖ It does not allow making an efficient implementation if dynamically change of vertices number is required. Adding a new vertex can be done in $O(1)$, but removal results in $O(E)$ complexity.

12.2.3 Lists vs Matrices

The representation of a graph is selected depending upon the purpose of the graph. If the graph is sparse, i.e. there are not many edges, then the matrix will take up a lot of space, but the adjacency list does not have that problem, because it only keeps track of which edges are actually in the graph. On the other hand, if there are a lot of edges in the graph, or if it is fully connected, then the list requires more space because of all of the references needed.

If it is to be found out whether or not an edge exists between two vertices, it can be found out directly in an adjacency matrix. Whereas, a long linked list has to be traversed in order to know that it is not in the graph. On the other hand, if all of a vertex's neighbors are to be looked at, then the entire matrix will have to be scanned. Whereas, in the list, only the linked list of neighbors is scanned.

In matrices, it is difficult to insert a new vertex or delete an existing vertex. For insertion and deletion, the size of the matrix has to be changed and all the vertices may have to be reordered. Thus, a lot of work has to be performed to carry out these operations. However, these operations can easily and properly be managed with adjacency lists.

12.3 GRAPH TRAVERSAL

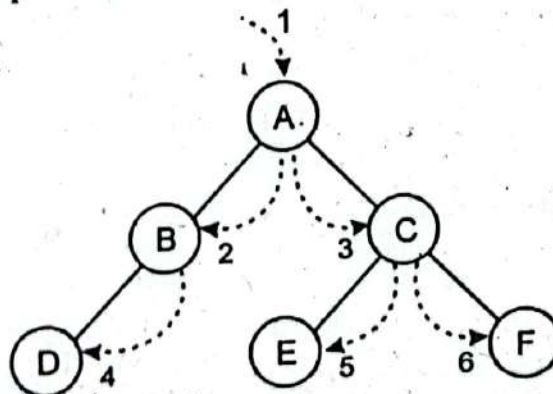
Graph traversal means visiting all the vertices or nodes of the graph. This operation on the graph is similar to the tree traversal. Graph traversal is the technique used for searching a vertex in a graph. Graph traversal is also used to decide the order of vertices to be visited in a search process. There are two standard graph traversal techniques. These are as follows:

- i) Breadth-First Search (BFS)
- ii) Depth-First Search (DFS)

12.3.1 Breadth-First Search (BFS)

Breadth-First Search (BFS) is the most commonly used algorithm for traversing or searching a graph. It traverses the vertices of a graph breadth-wise. Suppose the search process starts at some vertex 'v'. After visiting 'v', it visits all its neighbors (that has edges from 'v'), then unvisited neighbors of the neighbors, etc. This process continues until all the vertices in the graph are visited. Recursion is not used in BFS because traversing is done one level at a time. A queue data structure is generally used to implement the BFS algorithm.

For example, an undirected graph is given below which has 6 vertices. Suppose we start the traversal process from vertex A. After visiting this vertex, we will see its adjacent vertices. It has two adjacent vertices which are B and C. We will visit these unvisited vertices, from left to right. In this case, we will first visit vertex B and then C. Now we will visit D, then E and F. So all the vertices of the given graph have been visited.



Output: A, B, C, D, E, F

Time Complexity of BFS

The time complexity of BFS depends on the data structure used to represent the graph. If it is an adjacency matrix, it will be $O(V^2)$, and if it is an adjacency list, it will be $O(V+E)$, where V is the number of nodes and E is the number of edges.

Algorithm - Breadth-First Search

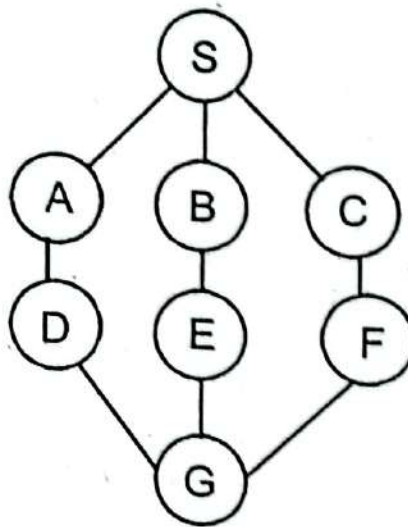
Write an algorithm that explains the Breadth-First Search.

1. START
2. Define a Queue with the size of the total number of vertices in the graph.

3. Select any vertex as a starting point for traversal. Visit that vertex and insert it into the Queue.
4. Visit all the adjacent vertices of the vertex which is in front of the Queue. Insert the vertices into the queue which are not visited.
5. Check the unvisited vertices that are adjacent to the vertex at front of the Queue. If there are no unvisited vertices adjacent to that vertex then delete it from the queue.
6. Repeat steps 4 and 5 until the queue becomes empty.
7. EXIT

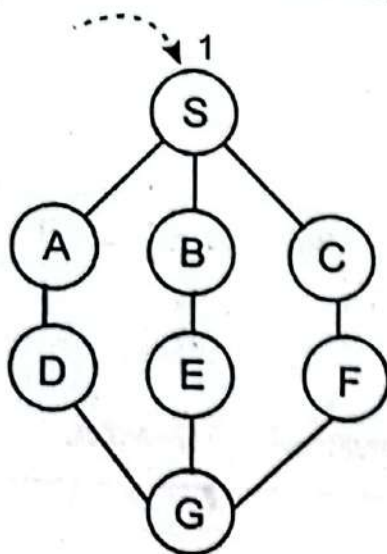
Example:

Consider the following graph:

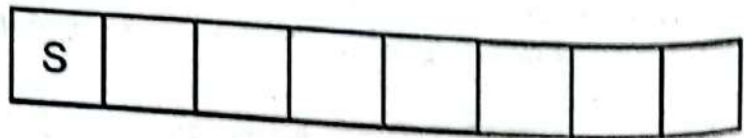


Following steps are performed to visit the above graph using the Breadth-First Search (BFS) technique:

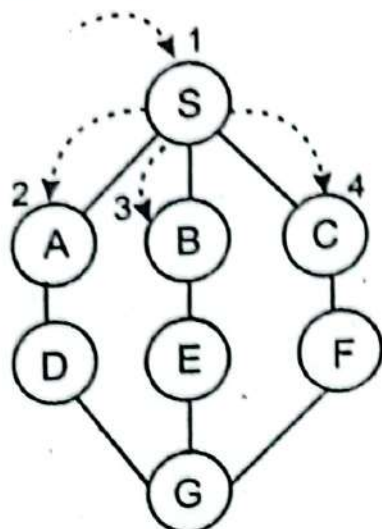
- 1- Initialize the queue.
- 2- Select the Vertex S as a starting point and visit it. Insert S into the queue.



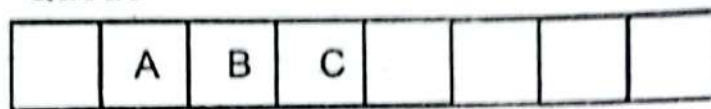
Queue



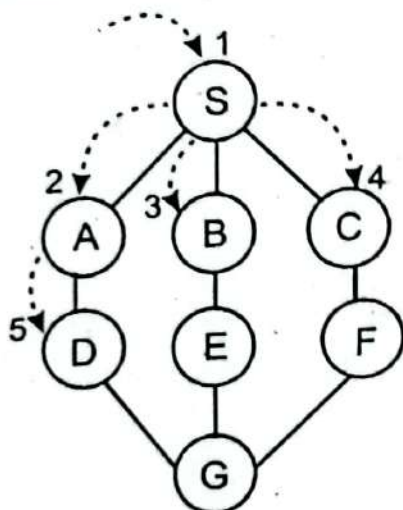
- 3- Visit all adjacent vertices of S which are not visited. These vertices are A, B and C. Insert these vertices into the queue and then delete S from the queue.



Queue



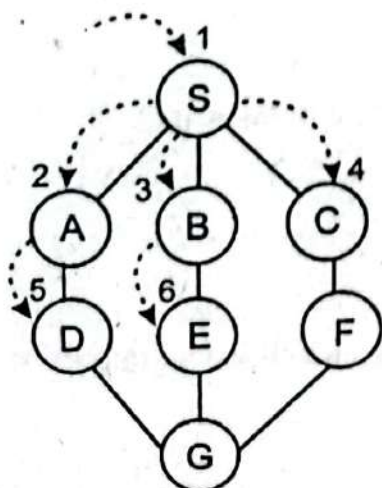
- 4- Visit all adjacent vertices of A which are not visited. There is only one vertex, which is D. Insert this vertex into the queue and then delete vertex A from the queue.



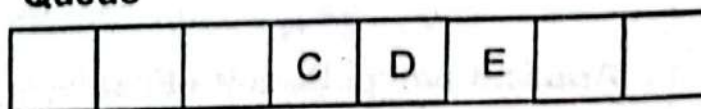
Queue



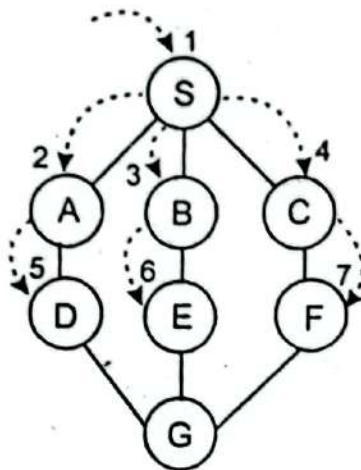
- 5- Visit all adjacent vertices of B which are not visited. There is only one vertex, which is E. Insert this vertex into the queue and then delete vertex B from the queue.



Queue



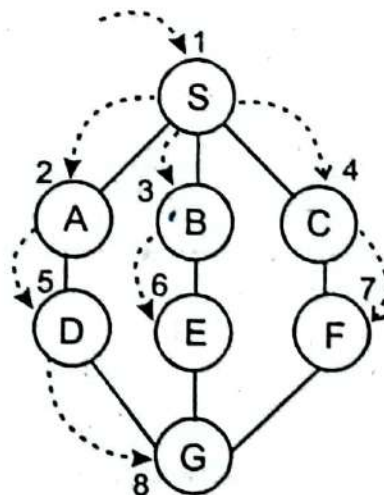
- 6- Visit all adjacent vertices of C which are not visited. There is only one vertex, which is F. Insert this vertex into the queue and then delete vertex C from the queue.



Queue

				D	E	F	
--	--	--	--	---	---	---	--

- 7- Visit all adjacent vertices of D which are not visited. There is only one vertex, which is G. Insert this vertex into the queue and then delete vertex D from the queue.



Queue

					E	F	G
--	--	--	--	--	---	---	---

- 8- Now there are three entries of the vertices in the queues. These vertices have no unvisited adjacent vertices. Delete these vertices from the queue.
- 9- Queue became empty. So, stop the BFS process.
- 10- The output of BFS for the above graph will be as under:

S, A, B, C, D, E, F, G

12.3.1.1 Applications of Breadth-First Search

There are many applications of Breadth-First Search; some important applications are as follows:

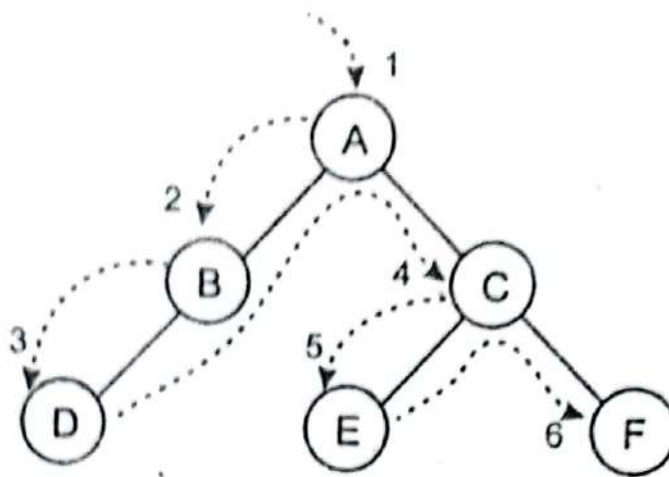
- 1- **Finding Shortest Path:** BFS is generally used for finding the shortest path in a graph and solving puzzle games.
- 2- **Peer-to-Peer Networks:** In peer-to-peer networks, it is used to find all neighbor nodes.
- 3- **Crawlers in Search Engines:** Crawlers build an index using the Breadth-First Search technique. The idea is to start from the source page and follow all links from the source and keep doing the same. However, Depth First Search can also be used for crawlers.
- 3- **Social Networking Websites:** In social networks, we can find people within a given distance 'k' from a person using Breadth-First Search till 'k' levels.
- 4- **GPS Navigation Systems:** Breadth-First Search is used to find all neighboring locations.
- 5- **Broadcasting in Network:** In networks, a broadcasted packet follows Breadth-First Search to reach all nodes.
- 6- **Cycle Detection in Undirected Graph:** In undirected graphs, Breadth-First Search can be used to detect a cycle. However, in a directed graph, only Depth First Search can be used.

12.3.2 Depth-First Search (DFS)

Depth First Search (DFS) is another way of traversing a graph. It is similar to the preorder traversal of a tree. It traverses the depth of any particular path before exploring its breadth. It is a recursive algorithm. A stack is generally used when implementing this algorithm.

The traversal process can be started from any vertex in the graph. Suppose a vertex 'v' is selected, processed, and marked. Then all unmarked vertices adjacent to 'v' are traversed. We go as deep as possible from neighbor to neighbor in a single path. When there is no more vertex to visit in a path, we trackback or take a U-turn and visit the next vertices. We will repeat the process until all vertices of the graph have been visited.

For example, an undirected graph is given below which has 6 vertices. Suppose we start the traversal process from vertex A. After visiting this vertex, we will see its adjacent vertices. It has two adjacent vertices which are B and C. We always select the next adjacent unvisited vertex in ascending order (or alphabetical order). In this case, we will select vertex B (not C). After visiting vertex B, we will see its adjacent vertices. It has one adjacent vertex which is D. We will visit this vertex i.e. vertex D. Now we will take a U-turn and go back to vertex A and see its next unvisited vertices which is only one i.e. vertex C. We will visit vertex C and then we will see its adjacent vertices. It has two adjacent vertices which are E and F. We will visit vertex E and then F. Now all the vertices of the given graph have been visited.



Output: A, B, D, C, E, F

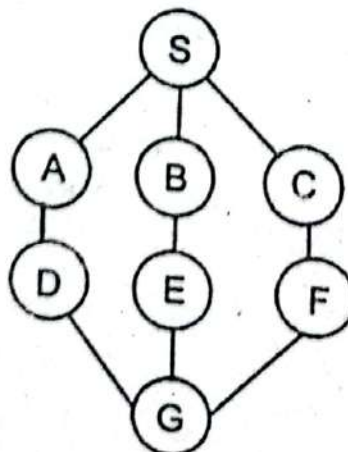
Algorithm – Depth-First Search (DFS)

Write an algorithm to explain the Depth-First Search.

1. START
2. Define a Stack with the size of the total number of vertices in the graph.
3. Select any vertex as a starting point for traversal. Visit that vertex and push it onto the Stack.
4. Visit all the adjacent vertices of the vertex which is at the top of the Stack. Push the vertices onto the Stack which are not visited.
5. Repeat step 4 until there is no new vertex to visit from the vertex on top of the Stack.
6. When there is no new vertex to visit then use backtracking and pop one vertex from the Stack.
7. Repeat steps 4 to 6 until Stack becomes Empty.
8. EXIT

Example:

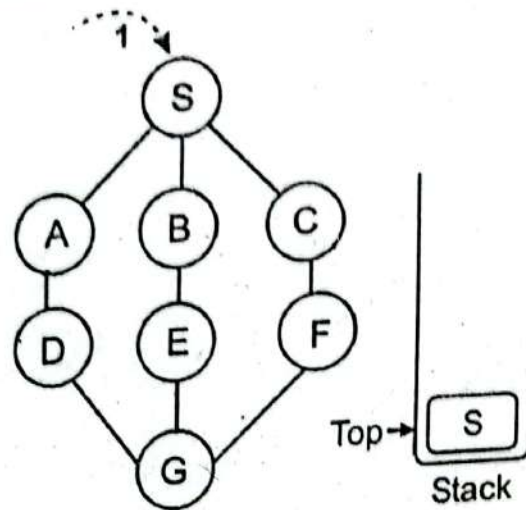
Consider the following graph:



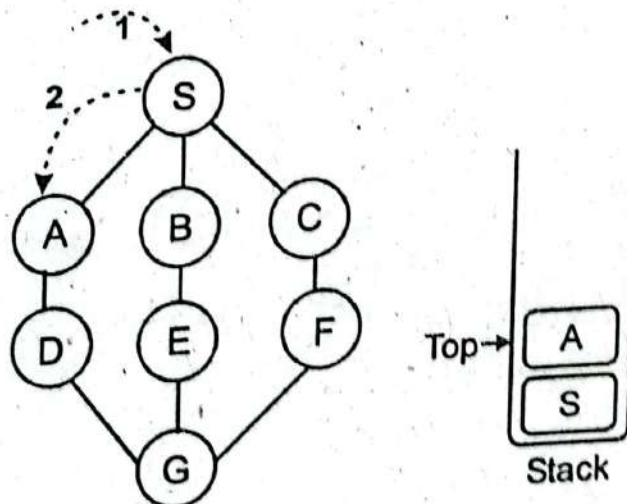
Following steps will be performed for traversal of the above graph using Depth First Search (DFS):

1- Initialize the Stack.

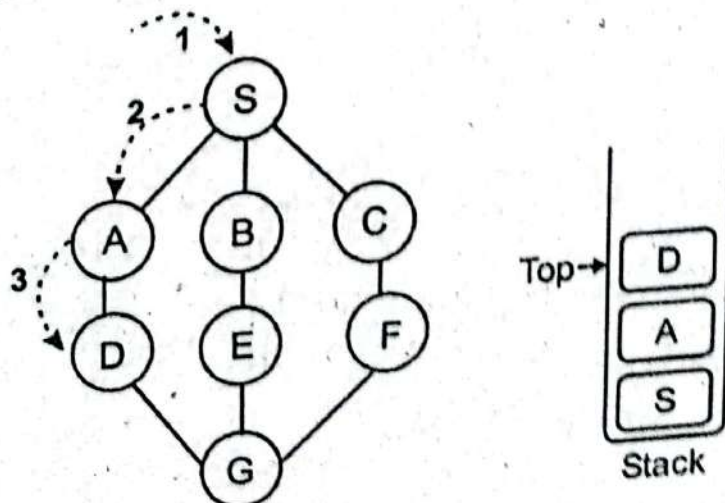
2- Select the Vertex S as a starting point and visit it. Push S onto the Stack.



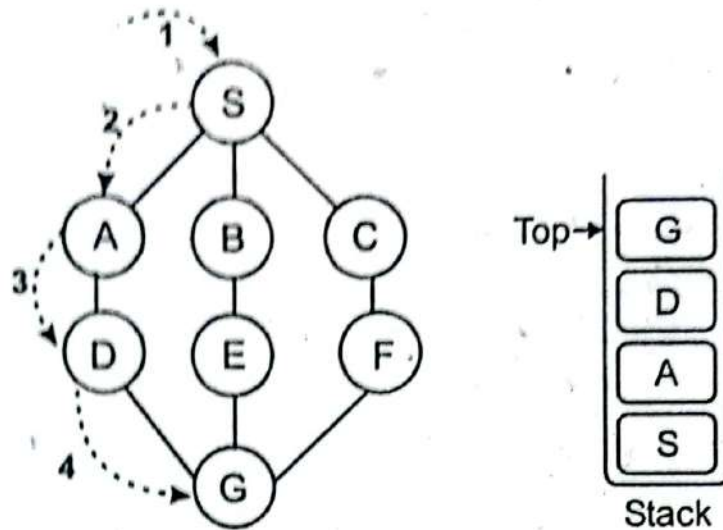
3- Look at adjacent vertices of S which are not visited. We have three vertices and we can pick any one of them. For this example, we shall take the vertex in alphabetical order (or ascending order). We select A and put it onto the Stack.



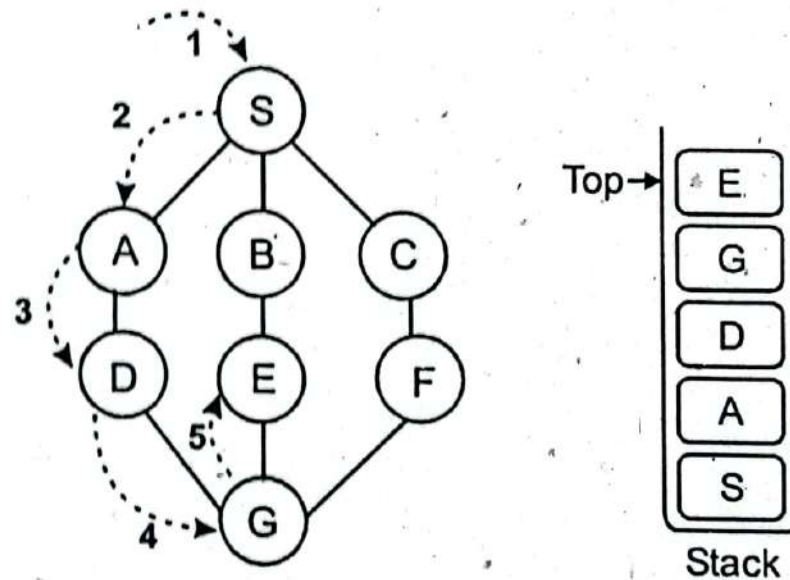
4- We select D and put it onto the Stack because this vertex is adjacent to A.



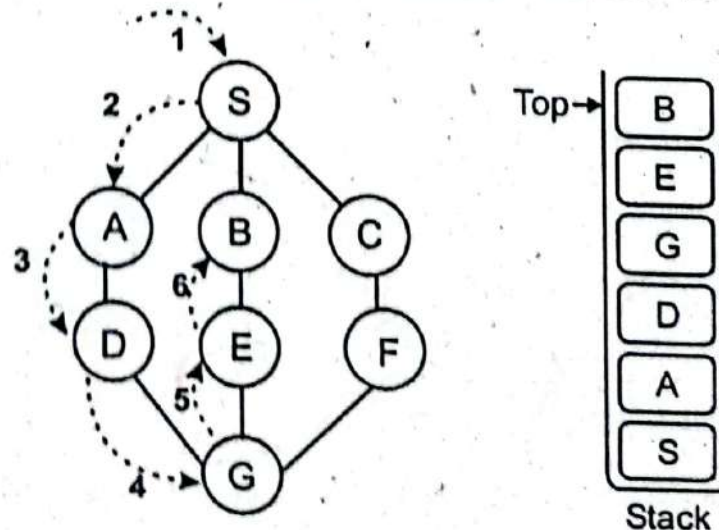
5. We select vertex **G** and put it onto the Stack because this vertex is adjacent to **D**.



6. Here, we have **E** and **F** vertices, which are adjacent to **G** and both are unvisited. However, we shall again choose in alphabetical order. We select **E** and put it onto the Stack.

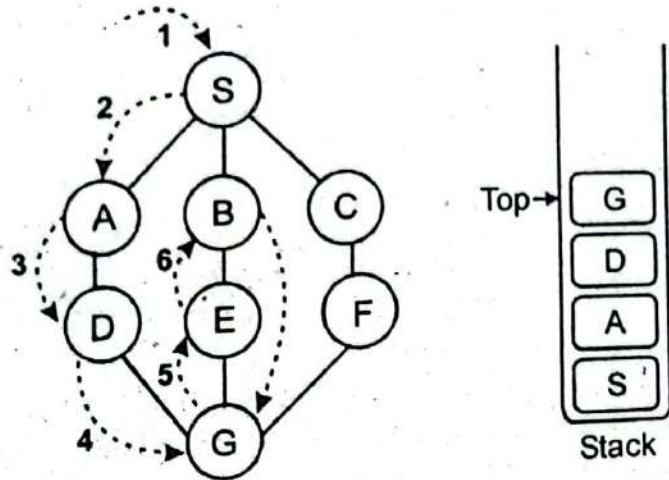


7. We select vertex **B** and put it onto the Stack because this vertex is adjacent to **E**.



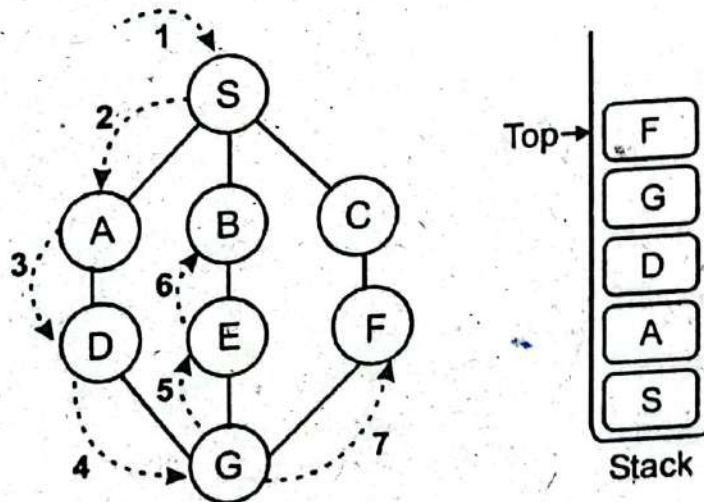
8-

The vertex **B** at the top of the Stack does not have any unvisited adjacent vertex. So, we pop **B** from the Stack. Similarly, the next top of the vertex in Stack is **E**. It also does not have any unvisited adjacent vertex. So, we pop **E** from the Stack. After popping vertices **B** and **E** from Stack, the Stack position is shown in the figure below



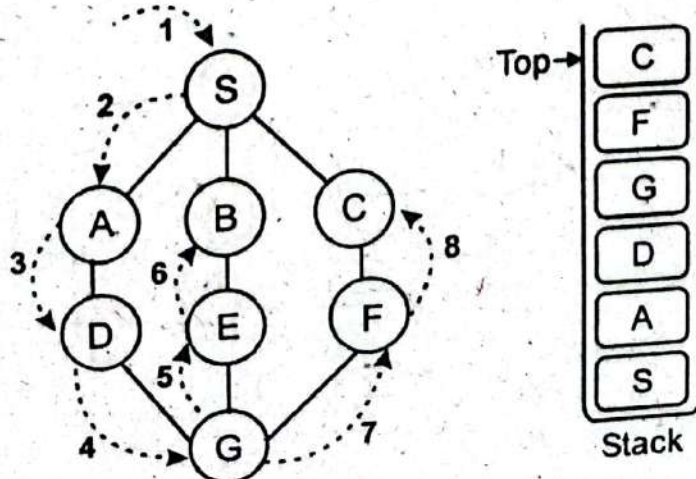
9-

The vertex **G** at the top of the Stack has one unvisited adjacent vertex which is **F**. So push **F** onto the Stack.



10-

The vertex **F** at the top of the Stack has one unvisited adjacent vertex which is **C**. So push **C** onto the Stack.



- 11- The vertices C, F, G, D, A, and S do not have any unvisited adjacent vertex, so we pop the Stack until it is empty.
- 12- Therefore, the output of DFS for the above graph will be as under:
S, A, D, G, E, B, F, C

12.3.2.1 Applications of Depth First Search

There are many applications of Depth First Search; some important applications are as follows:

- 1- **Path finding:** We can use the DFS algorithm to find a path between two given vertices such as V1 and V2.
- 2- **Topological sorting:** Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when re-computing formula values in spreadsheets, etc.
- 3- **Finding strongly connected components of a graph:** A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex.
- 4- **Solving puzzles with only one solution, such as mazes:** DFS can be used to find all solutions to a maze by only including vertices or nodes on the current path in the visited set.

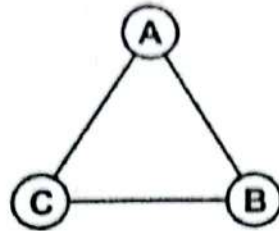
12.3.3 Difference between BFS & DFS

Following table shows the main differences between Breadth-First Search (BFS) and Depth First Search (DFS):

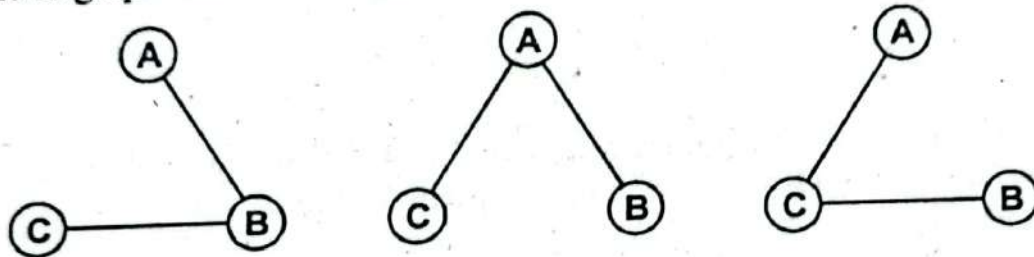
Sr#	BFS	DFS
1.	BFS is useful in finding the shortest path. It finds the shortest path to the destination.	DFS is not so useful in finding the shortest path. It goes to the deepest path in searching the vertices and then trackbacks.
2.	BFS requires more memory than DFS.	DFS requires less memory than BFS.
3.	BFS is slower than DFS.	DFS is faster than BFS.
4.	BFS is implemented using a queue (FIFO) data structure.	DFS is implemented using a stack (LIFO) data structure.
5.	BFS is a non-recursive algorithm.	DFS is a recursive algorithm.

12.4 **SPANNING TREE**

The spanning tree of undirected graph G is the sub-graph of G that is a tree with no cycles and contains all the vertices of G . Consider a connected undirected graph G with three vertices (A, B, C) and three edges (AB, CB, BA) as shown in the following figure:

**Graph G**

Following figures show three spanning trees of the graph G . Note that each tree has the same number of vertices and edges. However, the number of vertices or nodes of these spanning trees is the same as graph G but one edge is less than graph G .

**Spanning Trees****Properties of Spanning Tree**

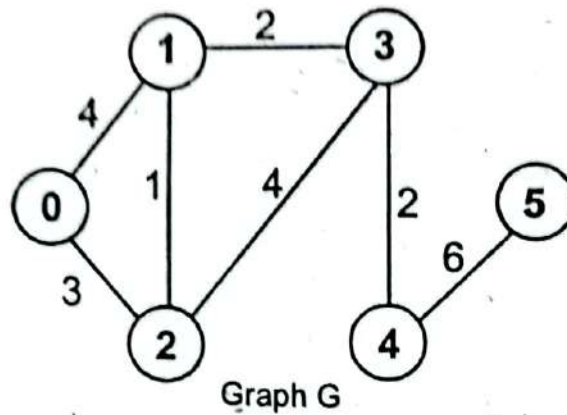
Following are some properties or characteristics of a spanning tree:

1. Every connected undirected graph G has at least one spanning tree. A disconnected graph does not have any spanning tree.
2. A connected graph G can have more than one spanning trees. For example, in the above graph G , there are three spanning trees.
3. A complete undirected graph G can have n^{n-2} number of spanning trees, where n is the number of vertices of the graph. In the above figure, graph G has 3 vertices, hence $3 (3^{3-2} = 3)$ spanning trees are possible to construct.
4. All the spanning trees of graph G have the same number of vertices and edges.
5. A spanning tree has the same number of vertices as graph G .
6. A spanning tree has $n-1$ edges, where n is the number of vertices. In the above example, three spanning trees of graph G are given. Each spanning tree has three vertices and two edges.
7. Spanning trees do not have cycles or loops, and cannot be disconnected. Adding one edge to the spanning tree will create a loop. Similarly, removing one edge from a spanning tree will make the spanning tree disconnected.
8. From a complete graph, if a maximum of $e-n+1$ edges are removed, a spanning tree can be constructed for that graph.

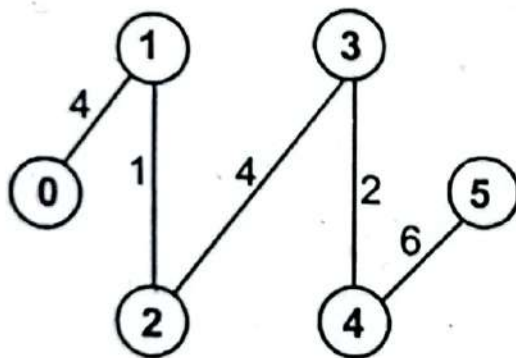
12.4.1 Minimum Spanning Tree (MST)

The cost of the spanning tree is the sum of the weights of all its edges. There can be many spanning trees of a weighted graph G . The spanning tree of a connected weighted graph G that has a minimum cost than all other spanning trees of the same graph is known as the *minimum spanning tree*. It is also called a *minimum weight spanning tree*.

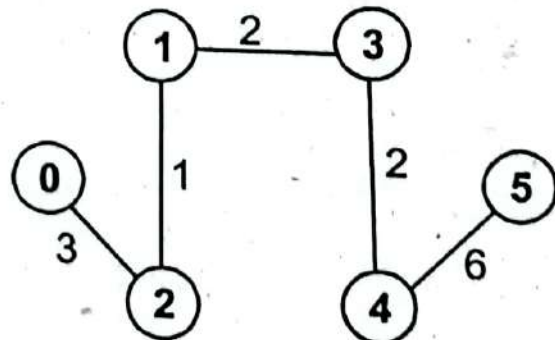
Consider a connected undirected weighted graph G with six vertices and seven edges as shown in the figure below.



This undirected weighted graph can have 16 ($4^{4-2} = 16$) possible spanning trees having their own costs. Two of them are shown in the following figures:



(a) Spanning Tree
Cost = $4+1+4+2+6 = 17$



(b) Spanning Tree
Cost = $3+1+2+2+6 = 14$

The spanning tree shown in figure (a) has a cost equal to 17 while the spanning tree shown in figure (b) has a cost equal to 14. The spanning tree with cost 14 is considered to be the minimum spanning tree.

There can be more than one minimum spanning tree of the same graph (In situation, when most of the edges of the graph have the same weights). If all the edges of the weighted graph have the same weights, then every spanning tree of that graph is a minimum spanning tree. However, if each edge of the weighted graph has a distinct weight then there will be only one, unique minimum spanning tree.

Applications of Minimum Spanning Tree

The spanning tree is basically used to find the minimum path to connect nodes in a graph. Some of the applications of minimum spanning tree are as follows:

1. **Internet routing:** Spanning trees are often used for Internet routing algorithms. On the Internet, computers (nodes) are often connected with many redundant physical connections. For efficiency, a routing algorithm tries to create a spanning tree to minimize the path (number of edges) between the computers (nodes) in the network. Ethernet uses a Spanning Tree Protocol to establish a cycle-free network.
2. **Solving the wiring problem:** Minimum spanning trees can be used to solve a wiring problem of a house with minimum cables.
3. **Designing network:** Minimum spanning tree approach can be used to design networks like phone networks, computer networks, with a minimum total cost. It can also be used for civil network planning.
4. **Traveling Salesman Problem:** Travelling salesman problem is a problem in which a list of cities along with distances among the pair of cities is given to the salesman. He is asked to travel while choosing the shortest possible route that visits each city and returns to the origin city.

12.4.2 Minimum Spanning Tree Algorithms

There are two famous algorithms (both are greedy algorithms) for finding a minimum spanning tree:

- i) Kruskal's Algorithm
- ii) Prim's Algorithm

12.4.2.1 Kruskal's Algorithm

Kruskal's algorithm is used for finding a minimum cost spanning tree for a connected weighted graph. It was introduced by Joseph Bernard Kruskal (American mathematician and computer scientist) in 1956. Kruskal's algorithm uses the greedy approach and works on the smallest edge first technique. This algorithm treats the graph as a forest and every node of the graph is treated as an individual tree. Forest is a collection of trees. A tree connects to another only if it has the least cost among all available options and does not violate MST properties.

Before applying Kruskal's algorithm to the graph, all the loops and parallel edges are removed from the graph. It sorts all the edges of the weighted graph in ascending order (based on weights of edges). Then, it gets the edges in order, one by one. If the edge does not create a cycle, it adds that edge to the spanning tree. If any edge causes a cycle, Kruskal's algorithm discards that edge. In every step, Kruskal's algorithm creates a tree and by merging operation, the minimum spanning tree grows step by step. The algorithm terminates when all the edges have been examined or when $n-1$ edges (where n is the number of vertices of the graph) have been added to the tree i.e. minimum spanning tree is created. The outcome of Kruskal's algorithm is the minimum spanning tree of the input graph.

Time Complexity of Kruskal's Algorithm

In Kruskal's algorithm, the most consuming process is sorting. The overall time complexity of Kruskal's algorithm is $O(E \log V)$, where E is the number of edges in the graph and V is the number of vertices.

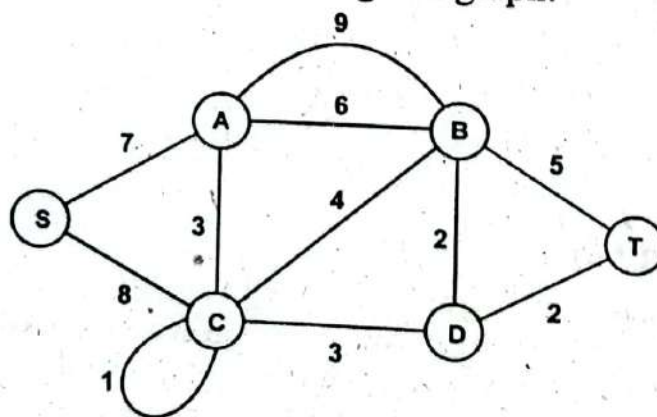
Algorithm – Kruskal's Algorithm

Write an algorithm that explains the Kruskal's Algorithm.

1. START
2. Remove all the loops and parallel edges. In the case of parallel edges, select (or keep) the one which has the least cost or weight associated and removes all other edges.
3. Define a Queue of the size of the total number of edges in the graph.
4. Sort all the edges from low weight to high and store them in a defined queue.
5. Select the edge with the lowest weight from the defined queue (from the rear of the queue).
6. If the selected edge creates a cycle, then reject this edge. Otherwise, add it to the minimum spanning tree and remove it from the queue.
7. Repeat steps 5 and 6 until all the edges are visited or the queue becomes empty.
8. EXIT

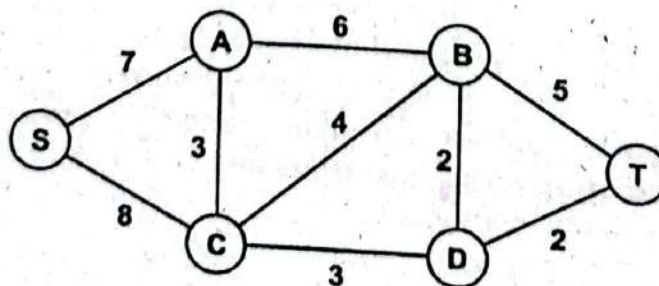
Example:

Consider the following undirected weighted graph:



Following steps are performed for finding the minimum spanning tree from the above graph using Kruskal's algorithm:

- 1- There is a loop as well as a parallel edge in the given graph. A loop exists at vertex C, and a parallel edge exists from A to B. Remove loop, and the parallel edge having maximum weight i.e. 9.

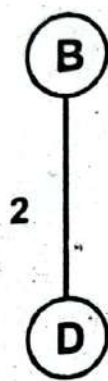


- 2- Declare a Queue of size 9 because the graph has nine edges.
- 3- Sort all the edges based on weights in ascending order and store them in a defined queue.

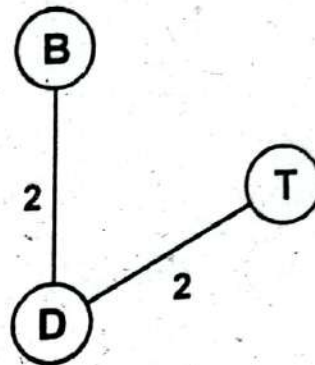
Edges	B-D	D-T	A-C	C-D	C-B	B-T	A-B	S-A	S-C
Weights	2	2	3	3	4	5	6	7	8

Queue

- 4- Select the edge with the least weight. In this case, the least weight is 2 and two edges B-D and D-T have this weight.
- Select B-D that is on the rear of the queue. It does not create a cycle. Add it to the minimum spanning tree (MST) and remove it from the queue.
 - Now D-T is on the rear of the queue. Select it. It also does not create a cycle. Add it to the MST and remove it from the queue.



Adding D-B

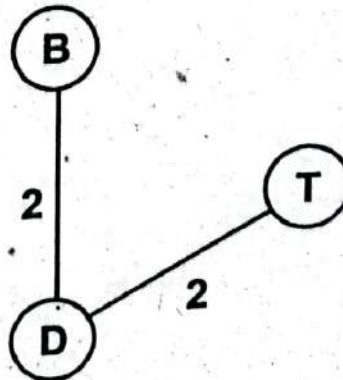
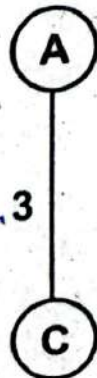


Adding D-T

Edges			A-C	C-D	C-B	B-T	A-B	S-A	S-C
Weights			3	3	4	5	6	7	8

Queue

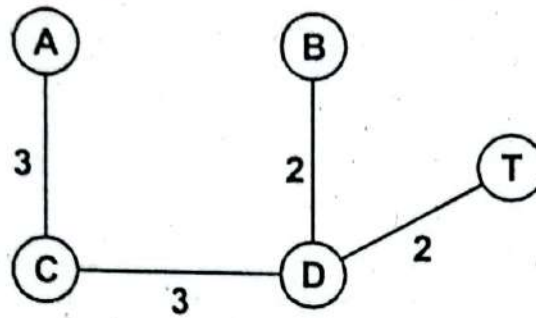
- 5- Select the next edge from the rear of the queue i.e. A-C. It does not create a cycle. Add it to the MST and remove it from the queue.



Edges				C-D	C-B	B-T	A-B	S-A	S-C
Weights				3	4	5	6	7	8

Queue

- 6- Select the next edge from the rear of the queue i.e. C-D. It does not create a cycle. Add it to the MST and remove it from the queue.



Edges					C-B	B-T	A-B	S-A	S-C
Weights					4	5	6	7	8
Queue									

- 7- Select the next edge from the rear of the queue i.e. C-B. It creates a cycle, rejects this edge, and also removes this edge from the queue. In this step, there would be no change in MST but the new status of the queue will be as under:

Edges						B-T	A-B	S-A	S-C
Weights						5	6	7	8
Queue									

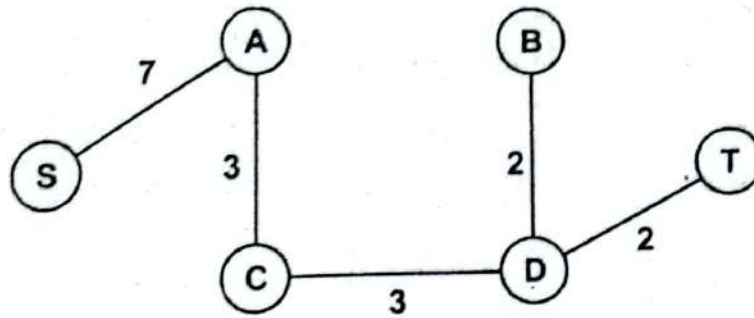
- 8- Select the next edge from the rear of the queue i.e. B-T. It creates a cycle, rejects this edge, and also removes this edge from the queue. In this step, there would be no change in MST but the new status of the queue will be as under:

Edges							A-B	S-A	S-C
Weights							6	7	8
Queue									

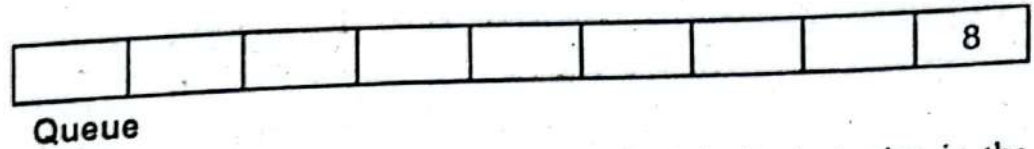
- 9- Select the next edge from the rear of the queue i.e. A-B. It creates a cycle, rejects this edge, and also removes this edge from the queue. In this step, there would be no change in MST but the new status of the queue will be as under:

Edges								S-A	S-C
Weights								7	8
Queue									

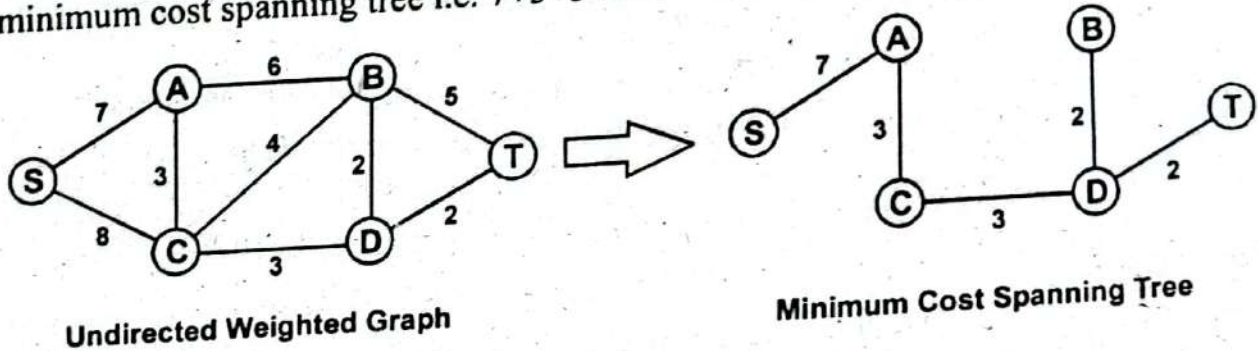
- 10- Select the next edge from the rear of the queue i.e. S-A. It does not create a cycle. Add it to the MST and remove it from the queue.



Edges
Weights



- 11- Select the next edge from the rear of the queue i.e. S-C. It is the last edge in the queue. It creates a cycle, rejects this edge, and also removes this edge from the queue. In this step, there would be no change in MST but the queue becomes empty.
- 12- Now all the edges of the graph have been visited, and we now have obtained the minimum cost spanning tree i.e. $7+3+3+2+2 = 17$. So, stop the process.



12.4.2.2 Prim's Algorithm

Prim's algorithm is another algorithm for finding a minimum cost spanning tree for a connected weighted graph. This algorithm was developed by Czech mathematician Vojtech Jarnik in 1930 and later rediscovered and republished by computer scientists Robert C. Prim in 1957, Dijkstra in 1959. This algorithm also uses a greedy approach and works on the nearest neighbor technique. In this method, one vertex is picked as root, and its neighborhood vertices are examined. We select the lowest weighted edge from the edges connected to the neighborhood vertices of the root and then add this edge to the MST (if it is valid for MST). Then take that neighborhood vertex as the root vertex and examined its associative edges and vertices. Repeat these steps until the minimum spanning tree is obtained.

Kruskal's algorithm makes forest (collection of trees) and by merging that forest, a minimum spanning tree is obtained. While Prim's algorithm creates a single tree and updates it in each stage.

Time Complexity of Prim's Algorithm

The time complexity of Prim's algorithm depends on the data structures used for the graph and for ordering the edges by weight, which can be done using a priority queue. Mostly

Prim's algorithm is implemented using an adjacency matrix or an adjacency list graph representation. In this case, the linearly searching of an array of weights is performed to find the minimum weight edge for adding to MST. This searching process requires $O(V^2)$ running time. However, this running time can be greatly improved further by using heaps for finding minimum-weight edges in the algorithm's inner loop.

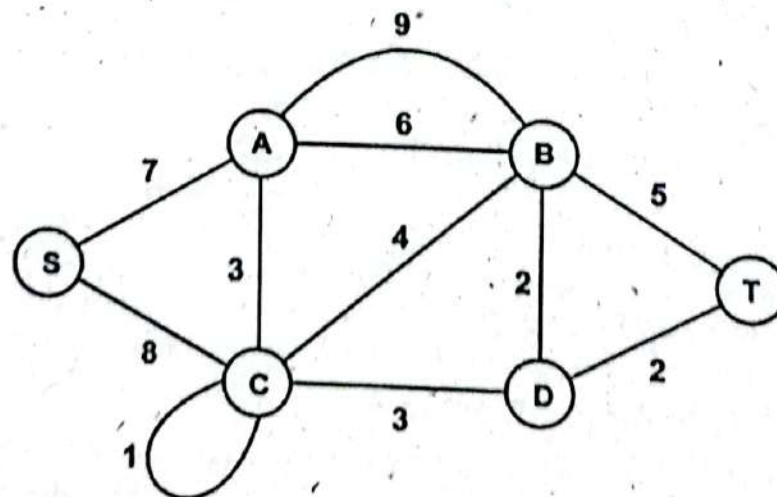
Algorithm – Prim's Algorithm

Write an algorithm that explains the Prim's Algorithm.

1. START
2. Remove all the loops and parallel edges. In the case of parallel edges, select (or keep) the one which has the least cost/weight associated and remove all other parallel edges.
3. Choose any arbitrary vertex as root vertex.
4. Check outgoing edges and select the one with the least cost.
5. If the selected edge creates a cycle, then ignore this edge and select another one. And if the selected edge is already a part of the MST, then ignore it. Otherwise, add it to the minimum spanning tree.
6. Change the root status to the vertex connected with the selected edge.
6. Repeat steps 4, 5, and 6 until all the vertices are visited.
8. EXIT

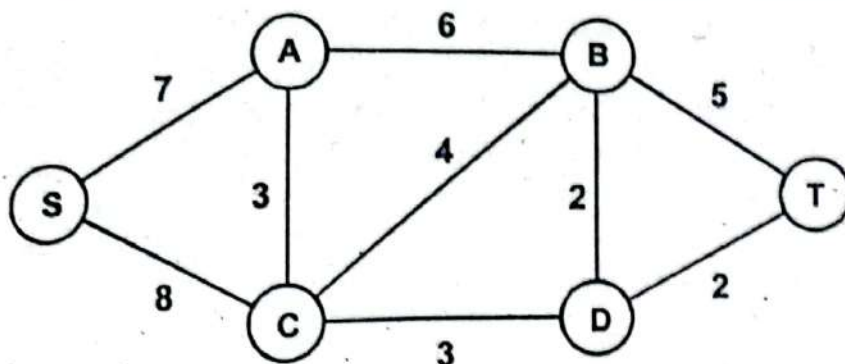
Example:

Consider the following undirected weighted graph:

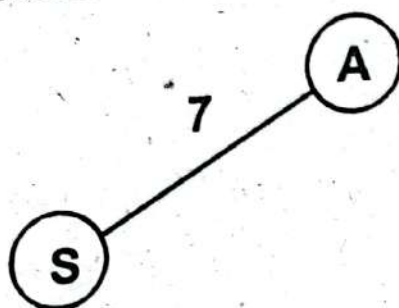


Following steps are performed for finding the minimum spanning tree from the above graph using Prim's algorithm:

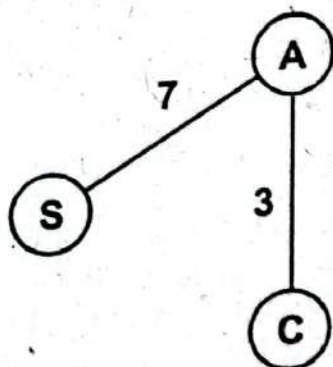
- 1- There is a loop as well as a parallel edge in the given graph. Loop is at vertex C, and a parallel edge exists from A to B. Remove loop, and the parallel edge having maximum weight i.e. 9.



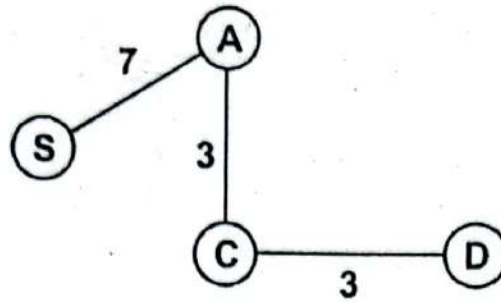
- 2- Choose arbitrary vertex as a root vertex/node. Suppose we choose S as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node.
- 3- The outgoing edges of S are S-A having cost 7, and S-C having cost 8. The edge S-A has the lowest cost among them. Hence, select this edge.
- 4- The selected edge S-A does not create any loop. So add it to MST.



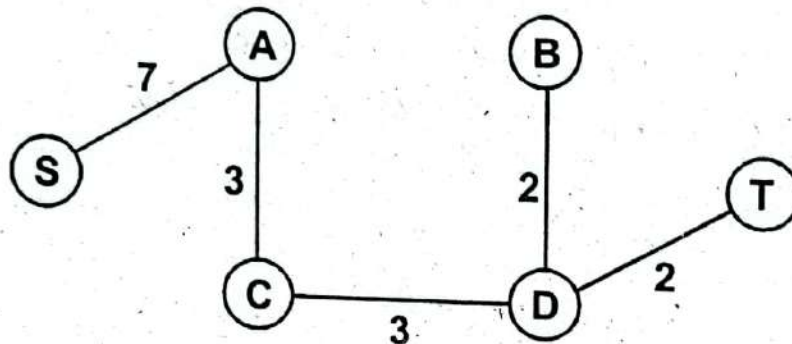
- 5- Change the root vertex. Now it is A.
- 6- The outgoing edges of A are: A-B having cost 6, A-C having cost 3, and A-S having cost 7 that is already an edge of MST. The edge A-C has the lowest cost which is 3. Hence, select this edge.
- 7- The selected edge A-C does not create any loop. So add it to MST.



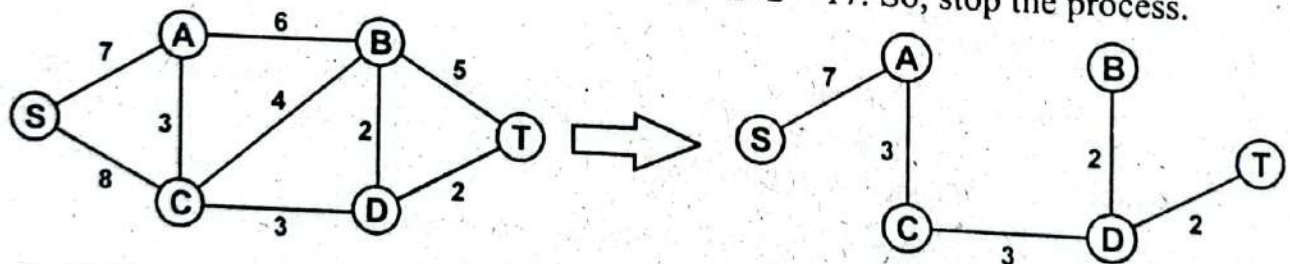
- 8- Change the root vertex to C.
- 9- The outgoing edges of C are: C-S having cost 8, C-A having cost 3, C-B having cost 4, and C-D having cost 3. The lowest cost is 3 of edges C-A and C-D. The edge C-A is already added to MST. Therefore, ignore edge C-A and select edge C-D for MST.
- 10- The selected edge C-D does not create any loop. So add it to MST.



- 11- Now change the root vertex to D.
- 12- The outgoing edges of D are: D-C having cost 3, D-B having cost 2, D-T having cost 2. The lowest cost is 2 of edges D-B and D-T and select them for MST.
- 13- Both of the selected edges are valid for MST (i.e. they do not create a loop). Thus, add them to MST.



- 14- Now all the vertices of the graph have been visited, and we now have obtained the minimum cost spanning tree i.e. $7+3+3+2+2 = 17$. So, stop the process.



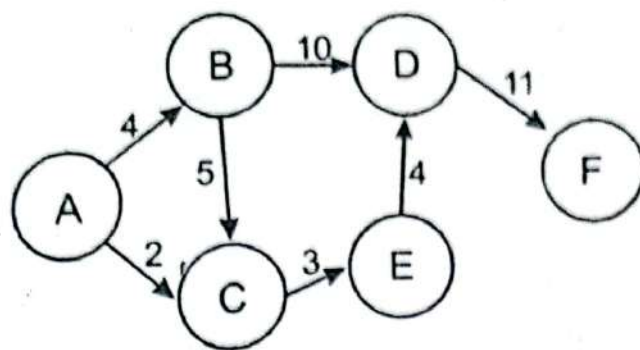
Undirected Weighted Graph

Minimum Cost Spanning Tree

12.5 SHORTEST PATH PROBLEM

The shortest path problem is about finding the shortest path (or lowest-cost path) between two vertices in a weighted graph. A path is said to be the shortest path in a graph if the sum of weights of all those edges that are participating (or existing) in a path, is minimum. The sum of weights of all the edges existing in the path between two vertices is known as the *length of the path*. An example of the shortest path problem is to find out the shortest path between two cities like finding the shortest path between Karachi and Islamabad. The shortest path problem is also known as a *single-pair shortest path problem*.

Consider the following directed graph:



In this graph, there are two possible paths between vertices A and F. First path is [A, B, D, F] and the second is [A, C, E, D, F]. The sum of the weights of the edges existing in the first path is 25 ($4+10+11 = 25$), while the sum of the weights of the edges existing in the second path is 20 ($2+3+4+11 = 20$). Hence, the second path [A, C, E, D, F] will be the shortest.

The difference between the *minimum spanning tree* and shortest path problem is that:

- The minimum spanning tree (MST) deals with the problems of connecting a set of all the vertices with the smallest cost.
- The shortest path problem only deals with the minimum cost path between the two vertices.

Applications of Shortest Path Problem

There are many applications of the shortest path problem in real-world applications. Some of them are as follows:

- **Mapping software:** It is used in mapping software for finding directions between physical locations such as driving directions on web mapping websites like Google map and Apple map. It is also used in vehicle navigation systems.
- **Internet packet routing:** It is used in Internet packet routing for transferring data packets to destination location via the shortest path.
- **Social network:** In the social network, it is used for finding the shortest path between two persons.
- **Video games:** In video games, it is used for finding the shortest path between two points on a map.

Types of Shortest Path Problem

There are three main types or variations of single-pair shortest path problem:

- 1- Single-source shortest path problem
- 2- Single-destination shortest path problem
- 3- All-pairs shortest path problem

12.5.1 Single-Source Shortest Path Problem

In this type/variation of the shortest path problem, the shortest path is determined from a single source vertex to all other vertices in the graph. Following are the popular algorithms which are used to solve the single-source shortest path problem:

- i) Dijkstra's Algorithm
- ii) Bellman-Ford Algorithm

12.5.1.1 Dijkstra's Algorithm

Dijkstra algorithm was introduced by a computer scientist Edsger W. Dijkstra in 1959. This algorithm is used to find out the shortest path from the source vertex to all other vertices in the given graph. It is a greedy algorithm. It can be used for both the directed graph and the undirected graph. All edges of the given graph must have non-negative weights. It means that Dijkstra's algorithm is only applicable to a graph that has edges of non-negative weights.

Dijkstra's algorithm applies relaxation upon vertices where needed. Relaxation means to update the cost if the current cost is smaller than the previous one. For a directed graph, use nodes with outgoing edges are used instead of neighboring vertices.

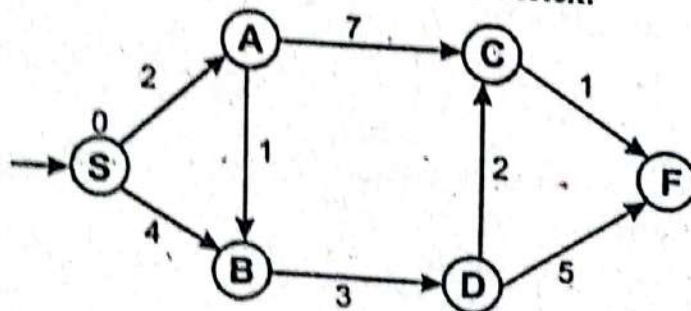
Algorithm – Dijkstra's Algorithm

Write an algorithm that explains the Dijkstra's Algorithm.

1. START
2. Set the cost of source vertex to 0 and all other vertices to infinity ∞ .
3. Insert them into the priority queue.
4. Select a vertex from the queue with the minimum distance or cost. If this vertex is already visited, then ignore it and select another vertex from the queue.
5. Calculate the costs of neighboring vertices of selected vertex. The formula of cost calculation is:
cost of the selected vertex + edge weight to a neighboring vertex
6. Apply relaxation upon neighboring vertices. That is if the calculated cost is less than the previous cost value, then update it with the calculated one.
7. Repeat steps 4 to 6 until all the vertices are visited or the queue becomes empty.
8. EXIT

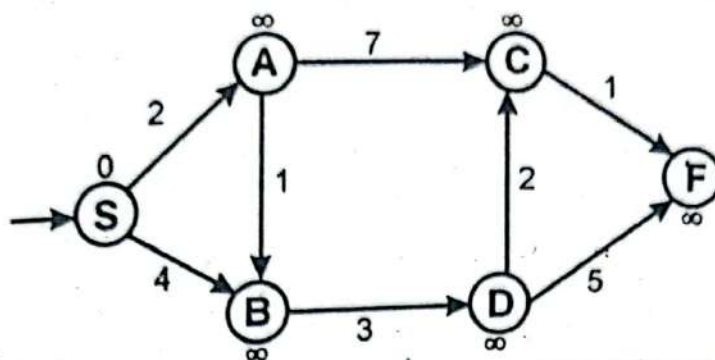
Example:

Consider the following graph. Let S be the source vertex.



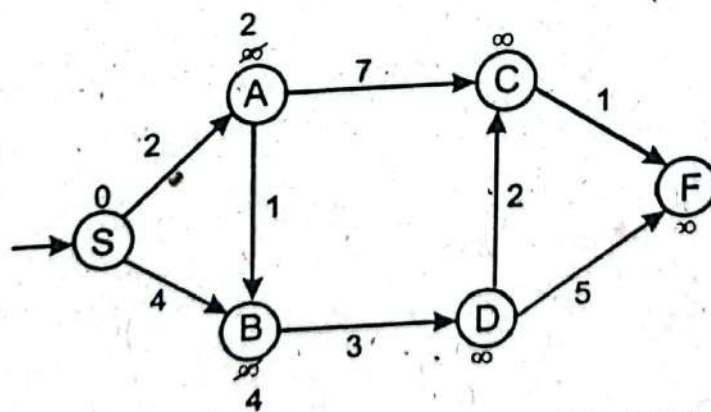
Following steps are performed for finding the shortest path from source vertex S to all other vertices in the graph using Dijkstra's algorithm:

- 1- Set the cost of source vertex i.e. S to 0, and the rest of all to infinity ∞ .



Visited Vertices	Cost of Vertices					
	S	A	B	C	D	F
{}	0	∞	∞	∞	∞	∞

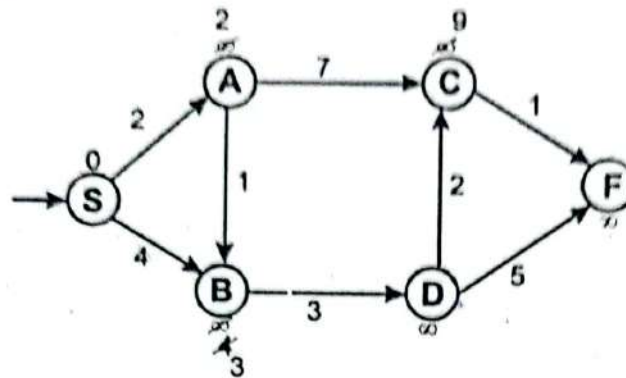
- 2- Select the vertex with minimum cost i.e. vertex S with 0 cost.
- 3- Calculate the distance/cost from vertex S to neighboring vertices i.e. S-A and S-B. Cost of neighbor A is $0 + 2 = 2$ and cost of vertex B is $0 + 4 = 4$.
- 4- Perform edge relaxation. For vertex A, the current cost is ∞ , and the calculated cost is 2. As $2 < \infty$, so update the cost of vertex A to 2. Similarly, for vertex B, the current cost is ∞ and the calculated cost is 4. As $4 < \infty$, so update the cost of vertex B to 4.



Visited Vertices	Cost of Vertices					
	S	A	B	C	D	F
{}	0	∞	∞	∞	∞	∞
{S}	0	2	4	∞	∞	∞

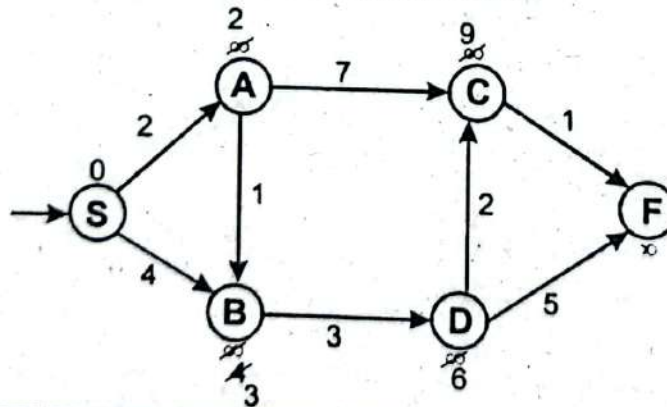
- 5- Now select the vertex with minimum cost. Vertex A has the minimum cost and not in the visited vertices list.
- 6- The neighbors of vertex A are vertices B and C. The cost of vertex B is $2 + 1 = 3$, while cost of vertex C is $2 + 7 = 9$.

- 7- Perform edge relaxation. For vertex B, the current cost is 4 and the calculated cost is 3. As $3 < 4$, so update the cost of vertex B to 3. For vertex C, the current cost is infinity and the calculated cost is 9 which is less than infinity. So, update the cost of vertex C to 9.



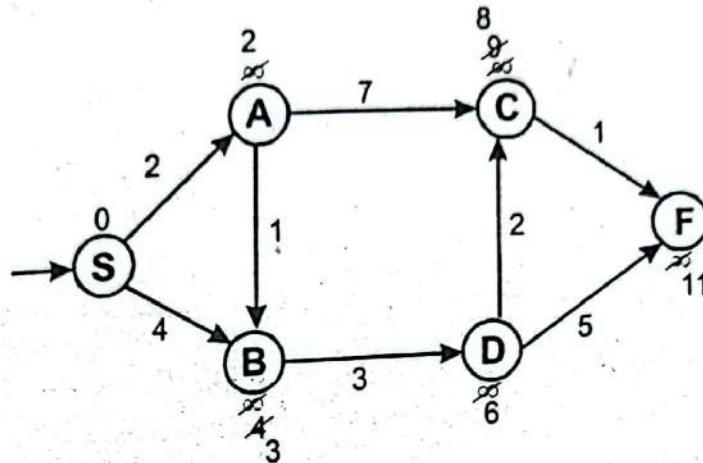
Visited Vertices	Cost of Vertices					
	S	A	B	C	D	F
{}	0	∞	∞	∞	∞	∞
{S}	0	2	4	∞	∞	∞
{S,A}	0	2	3	9	∞	∞

- 8- Now the minimum cost vertices S and A are already in the visited nodes list. So, select vertex B with cost 3.
- 9- The neighbor of vertex B is only D. So the cost of vertex D is $3 + 3 = 6$.
- 10- Perform edge relaxation. The calculated cost of vertex D is 6, which is less than infinity. Hence, update the cost of vertex D to 6.



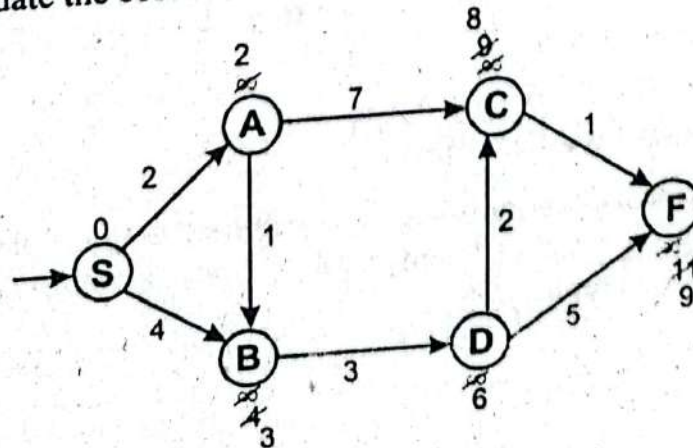
Visited Vertices	Cost of Vertices					
	S	A	B	C	D	F
{}	0	∞	∞	∞	∞	∞
{S}	0	2	4	∞	∞	∞
{S,A}	0	2	3	9	∞	∞
{S,A,B}	0	2	3	9	6	∞

- 11- Now the visited nodes are S, A and B. Select the next vertex with minimum cost. It is vertex D.
- 12- The neighbors of vertex D are vertices C and F. The cost of vertex C is $6+2 = 8$, while cost of vertex F is $6+5 = 11$.
- 13- Perform edge relaxation. For vertex C, the calculated cost 8 is less than the previous cost which is 9. So update it. Similarly, for vertex F, the calculated cost 11 is less than infinity. Update it also.



Visited Vertices	Cost of Vertices					
	S	A	B	C	D	F
{}	0	∞	∞	∞	∞	∞
{S}	0	2	4	∞	∞	∞
{S,A}	0	2	3	9	∞	∞
{S,A,B}	0	2	3	9	6	∞
{S,A,B,D}	0	2	3	8	6	11

- 14- Now all the vertices are visited except vertices C and F. The vertex C has less cost than F, so select C.
- 15- The vertex C has only one neighbor F. Calculate the cost of F from C i.e. $8+1 = 9$.
- 16- The previous value of vertex F is 11 and the calculated value is 9 which is less than 11. So, update the cost of vertex F.



Visited Vertices	Cost of Vertices					
	S	A	B	C	D	F
{}	0	∞	∞	∞	∞	∞
{S}	0	2	4	∞	∞	∞
{S,A}	0	2	3	9	∞	∞
{S,A,B}	0	2	3	9	6	∞
{S,A,B,D}	0	2	3	8	6	11
{S,A,B,D,C}	0	2	3	8	6	9

- 17- Now select vertex F. This vertex has no neighbors. Hence, add it to the visited vertices.

Visited Vertices	Cost of Vertices					
	S	A	B	C	D	F
{}	0	∞	∞	∞	∞	∞
{S}	0	2	4	∞	∞	∞
{S,A}	0	2	3	9	∞	∞
{S,A,B}	0	2	3	9	6	∞
{S,A,B,D}	0	2	3	8	6	11
{S,A,B,D,C}	0	2	3	8	6	9
{S,A,B,D,C,F}	0	2	3	8	6	9

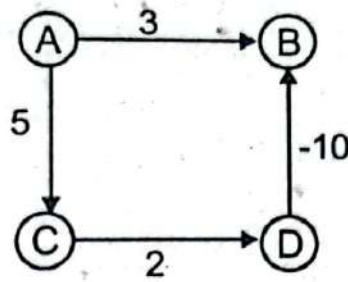
- 18- The output of Dijkstra's algorithm is the visited vertices. Thus, the shortest path is S-A-B-D-C-F.

Time Complexity of Dijkstra's Algorithm

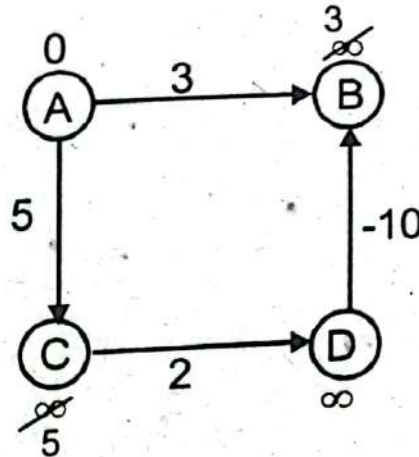
The time complexity of Dijkstra's algorithm is $O(V^2)$, where V is the number of vertices or nodes. But implementing it by min-priority queue it can be reduced to $O(V+E \log V)$, where E is the number of edges.

Drawback of Dijkstra's Algorithm

Dijkstra's algorithm works for non-negative weighted graphs only. For negative weights, this algorithm sometimes gives correct results and sometimes incorrect results. For example, consider the following directed graph:

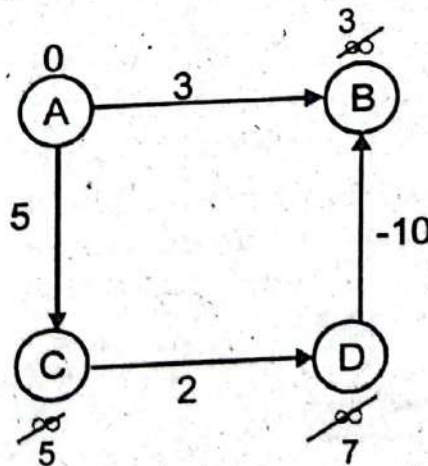


Let A be the source node. Set all the nodes initially to 0 or infinity ∞ . The vertex A is connected to vertices B and C. Calculate the cost and update them as described by Dijkstra's algorithm. The graph will look like:



Visited Vertices	Cost of Vertices			
	A	B	C	D
{}	0	∞	∞	∞
{A}	0	3	5	∞

Now minimum cost vertex is B. Select it, but it has no connection to any vertex, so add it to the visited vertices list. Select another vertex C, and apply the relaxation step as defined above.



Visited Vertices	Cost of Vertices			
	A	B	C	D
{}	0	∞	∞	∞
{A}	0	3	5	∞
{A, B}	0	3	5	7

{A}	0	3	5	∞
{A, B}	0	3	5	∞
{A, B, C}	0	3	5	7

Then select vertex D, its neighboring vertex is B. It is already visited, so by Dijkstra's algorithm, it cannot be relaxed again. But if we see the cost of vertex B for this iteration, it would be $7 + (-10) = -3$ which is less than the previous cost of B i.e. 3. If we try to update it at this stage, it could mix up all the previous costs. Dijkstra's algorithm works well for non-negative weights. But for negative weights, it is not a suitable approach.

12.5.1.2 Bellman-Ford Algorithm

Bellman-Ford algorithm is also used to find out the shortest path from the source vertex to all other vertices in the given graph (i.e. this algorithm also solves the single-source shortest path problem). The edge weights may be negative. It is similar to Dijkstra's algorithm but the main differences between these two algorithms are as follows:

- Dijkstra's algorithm does not work for graphs with negative weight edges, while the Bellman-Ford algorithm can work for such graphs. So, the Bellman-Ford algorithm is more versatile than Dijkstra's algorithm.
- Bellman-Ford algorithm is simpler than Dijkstra's algorithm.
- The time complexity of the Bellman-Ford algorithm is more than Dijkstra's algorithm. Hence, the Bellman-Ford algorithm is slower than Dijkstra's algorithm.
- Bellman-Ford algorithm can easily be implemented in distributed systems, while Dijkstra's algorithm cannot.

Bellman-Ford algorithm was first introduced by Alfonso Shimbel in 1955, but it was named after Richard Bellman and Lester Ford who published it in 1958 and 1956 respectively (Richard Bellman and Lester Ford were American mathematicians). Edward F. Moore also published the same algorithm in 1957. Therefore, it is also sometimes called the *Bellman-Ford-Moore algorithm*.

Negative weights can create negative weight cycles. Bellman-Ford algorithm can detect negative cycles. It finds out the shortest path if and only if no negative cycle exists. It relaxes all the edges progressively in $n-1$ iterations, where n is the number of vertices of the given graph. Sometimes, this algorithm can find the shortest path at the first iteration, or sometimes it takes multiple iterations to find the shortest path.

Algorithm – Bellman-Ford Algorithm

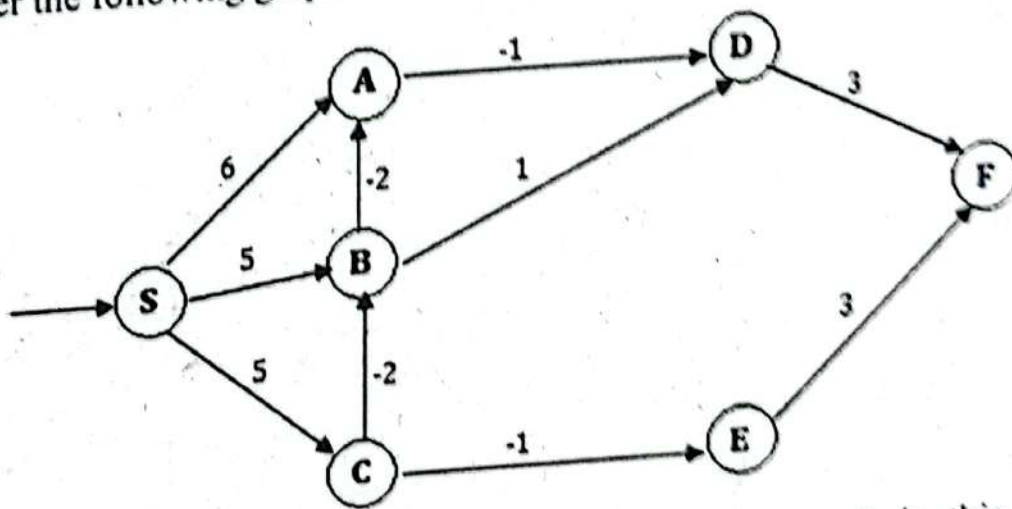
Write an algorithm that explains the Bellman-Ford algorithm.

1. START
2. Set the cost of source vertex to 0 and all other vertices to infinity ∞ .
3. Make a list of all edges in order.

4. Calculate the cost of the vertex which is at the edge head. The formula to calculate the cost is:
current cost of vertex + edge weight
5. Apply relaxation operation if calculated cost value is less than previous cost value, and then update it with calculated one.
6. Repeat steps 4 and 5 for all edges.
7. Repeat steps 4, 5, and 6 for $n-1$ times, where n is the total number of vertices of the given graph.
8. EXIT

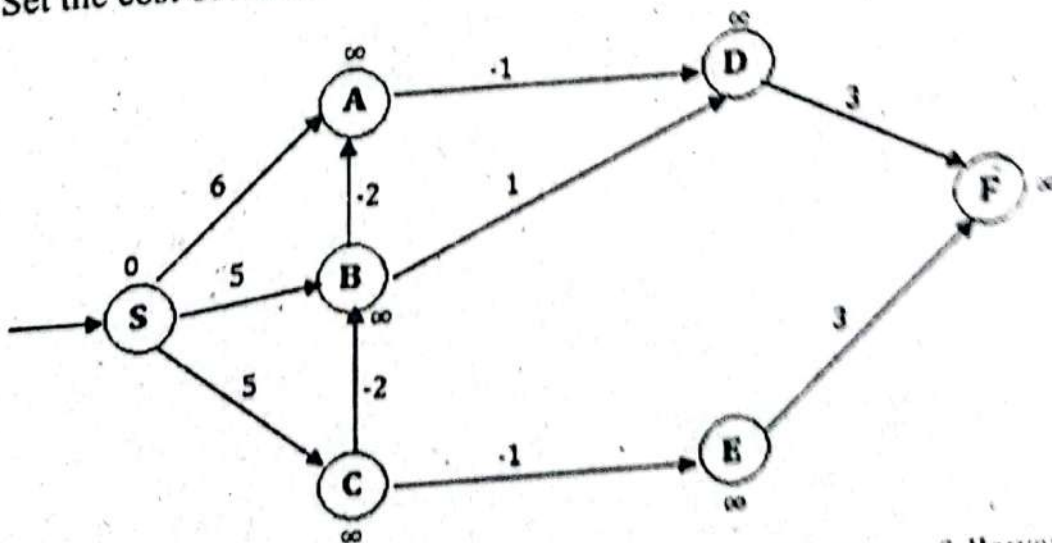
Example:

Consider the following graph. Let vertex S is the source vertex.



Following steps are performed for finding the shortest path in this graph using the Bellman-Ford algorithm:

- 1- Set the cost of source vertex i.e. S to 0, and all other to infinity ∞ .



- 2- List of edges in the given graph along with its weights are as follows:

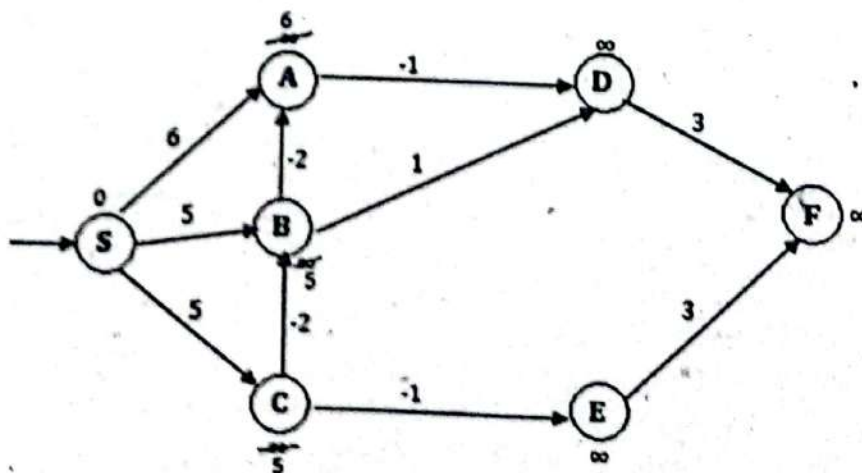
Edges	Weights
S-A	6
S-B	5
S-C	5
A-D	-1
B-A	-2
B-D	1
C-B	-2
C-E	-1
D-F	3
E-F	3

- 3- Relax all the edges progressively in 6 iterations (There are 10 edges, and 7 vertices. Thus, Bellman-Ford will iterate 6 times ($7-1=6$) for 10 edges each time). These iterations are described as under:

Iteration #1:

- i) For S-A; cost = 6, which is less than ∞ . Relax node A with value 6.
- ii) For S-B; cost = 5(0+5), which is also less than ∞ . Relax node B with value 5.
- iii) For S-C; cost = 5(0+5), which is also less than ∞ . Relax node C with value 5.
- iv) For A-D; cost = ∞ ($\infty+(-1)$). No change.
- v) For B-A; cost = ∞ ($\infty+(-2)$). No change.
- vi) For B-D; cost = ∞ ($\infty+1$). No change.
- vii) For C-B; cost = ∞ ($\infty+(-2)$). No change.
- viii) For C-E; cost = ∞ ($\infty+(-1)$). No change.
- ix) For D-F; cost = ∞ ($\infty+3$). No change.
- x) For E-F; cost = ∞ ($\infty+3$). No change.

The graph takes the following form:

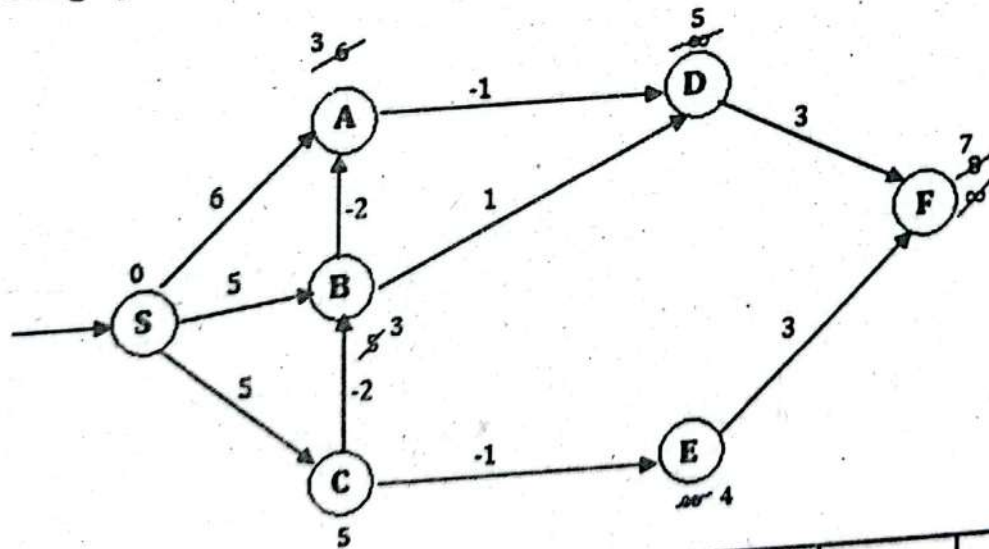


Iteration #	S	A	B	C	D	E	F
0	0	∞	∞	∞	∞	∞	∞
1	0	6	5	5	∞	∞	∞

Iteration #2:

- i) For S-A; cost: $0+6 = 6$, same as the previous one. No change.
- ii) For S-B; cost: $0+5 = 5$, same as the previous one. No change.
- iii) For S-C; cost: $0+5 = 5$, same as the previous one. No change.
- iv) For A-D; cost: $6+(-1) = 5$, which is less than ∞ . Relax node D with value 5.
- v) For B-A; cost: $5+(-2) = 3$, which is less than 6. Relax node A with value 3.
- vi) For B-D; cost: $5+1 = 6$. D has cost 5, and $5 < 6$. Hence, no change.
- vii) For C-B; cost: $5+(-2) = 3$, which is less than 5. Relax node B with value 3.
- viii) For C-E; cost: $5+(-1) = 4$, which is less than ∞ . Relax node E with value 4.
- ix) For D-F; cost: $5+3 = 8$, which is less than ∞ . Relax node F with value 8.
- x) For E-F; cost: $4+3 = 7$, which is less than 8. Relax node F with value 7.

The graph takes the following form:



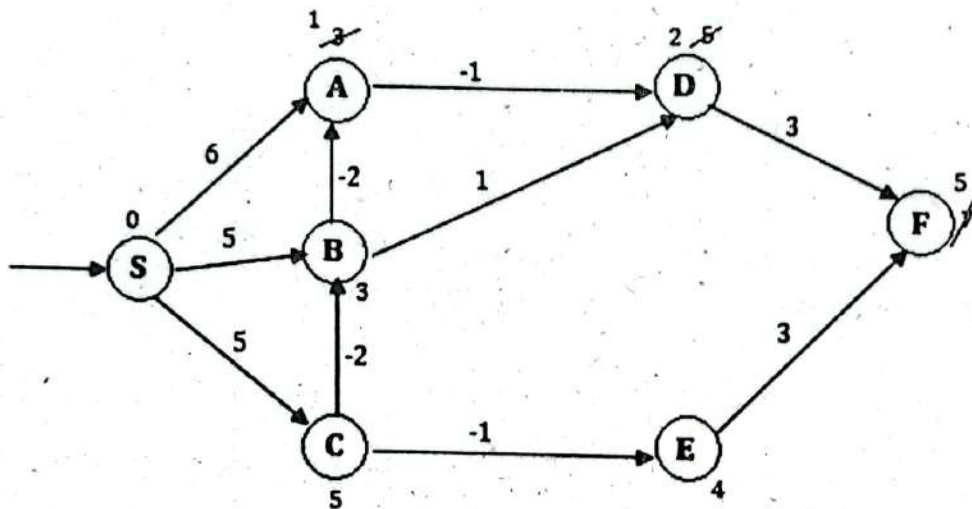
Iteration #	S	A	B	C	D	E	F
0	0	∞	∞	∞	∞	∞	∞
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	7

Iteration #3:

- i) For S-A; cost = $0+6 = 6$, $6 > 3$. No change.
- ii) For S-B; cost = $0+5 = 5$, $5 > 3$. No change.

- iii) For S-C; cost = $0+5 = 5$, same as previous one. No change.
- iv) For A-D; cost = $3+(-1) = 2$, which is less than 5. Relax node D with value 2.
- v) For B-A; cost = $3+(-2) = 1$, which is less than 3. Relax node A with value 1.
- vi) For B-D; cost = $3+1 = 4$. D has cost 2, and $2 < 4$. Hence, no change.
- vii) For C-B; cost = $5+(-2) = 3$, same as previous one. No change.
- viii) For C-E; cost = $5+(-1) = 4$, same as previous one. No change.
- ix) For D-F; cost = $2+3 = 5$, which is less than 7. Relax node F with value 5.
- x) For E-F; cost = $4+3 = 7$. F has cost 5, and $5 < 7$. Hence, no change.

The graph takes the following form:

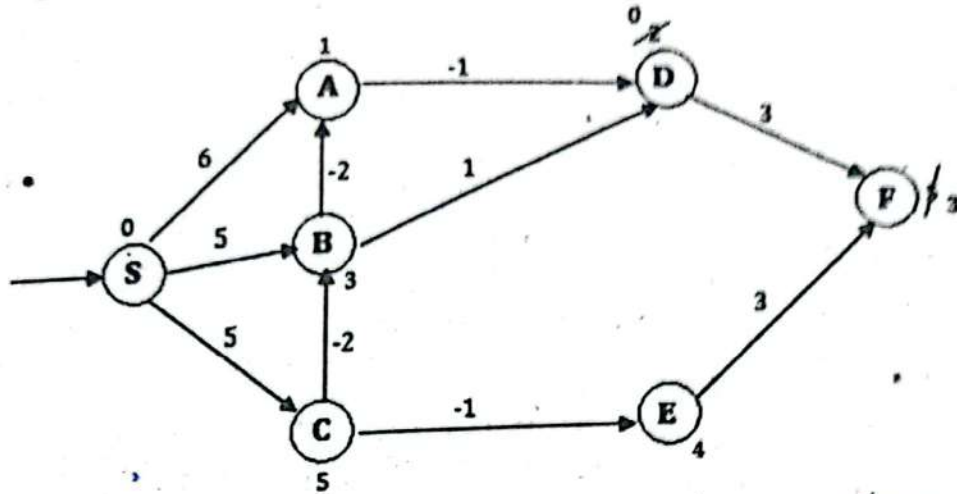


Iteration #	S	A	B	C	D	E	F
0	0	∞	∞	∞	∞	∞	∞
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	7
3	0	1	3	5	2	4	5

Iteration #4:

- i) For S-A; cost = $0+6 = 6$, $6 > 1$. No change.
- ii) For S-B; cost = $0+5 = 5$, $5 > 3$. No change.
- iii) For S-C; cost = $0+5 = 5$, same as previous one. No change.
- iv) For A-D; cost = $1+(-1) = 0$, which is less than 2. Relax node D with value 0.
- v) For B-A; cost = $3+(-2) = 1$, same as previous one. No change.
- vi) For B-D; cost = $3+1 = 4$. D has cost 2, and $2 < 4$. Hence, no change.
- vii) For C-B; cost = $5+(-2) = 3$, same as previous one. No change.
- viii) For C-E; cost = $5+(-1) = 4$, same as previous one. No change.

- ix) For D-F; cost = $0+3 = 3$, which is less than 5. Relax node F with value 3.
 x) For E-F; cost = $4+3 = 7$. F has cost 3, and $3 < 7$. Hence, no change.
 The graph takes the following form:



Iteration #	S	A	B	C	D	E	F
0	0	∞	∞	∞	∞	∞	∞
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	7
3	0	1	3	5	2	4	5
4	0	1	3	5	0	4	3

Iteration #5:

- i) For S-A; cost = $0+6 = 6$, $6 > 1$. No change.
 ii) For S-B; cost = $0+5 = 5$, $5 > 3$. No change.
 iii) For S-C; cost = $0+5 = 5$, same as previous one. No change.
 iv) For A-D; cost = $1+(-1) = 0$, same as previous one. No change.
 v) For B-A; cost = $3+(-2) = 1$, same as previous one. No change.
 vi) For B-D; cost = $3+1 = 4$. D has cost 0, and $0 < 4$. Hence, no change.
 vii) For C-B; cost = $5+(-2) = 3$, same as previous one. No change.
 viii) For C-E; cost = $5+(-1) = 4$, same as previous one. No change.
 ix) For D-F; cost = $0+3 = 3$, same as previous one. No change.
 x) For E-F; cost = $4+3 = 7$. F has cost 3, and $3 < 7$. Hence, no change.

No change in this iteration. So, the 6th iteration skipped here only for that case as the results obtained are the same as to the last iteration. The table shows the path lengths of each node.

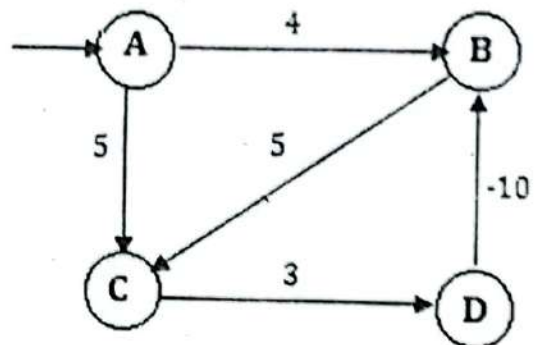
Time Complexity of Bellman-Ford Algorithm

The time complexity of the Bellman-Ford algorithm is $O(E.V)$, where V is the number of vertices and E is the number of edges of the given graph.

Drawback of Bellman-Ford Algorithm

Actually, Bellman-Ford should relaxed nodes in $n-1$ iterations only. One more iteration should not change the status of any vertex any more. But sometimes, Bellman-Ford relaxed vertices after $n-1$ iterations. For example for the given graph the iteration table is shown:

Iteration #	A	B	C	D
0	0	∞	∞	∞
1	0	-2	5	8
2	0	-4	3	6
3	0	-6	1	4
4	0	-8	-1	2
5	0	-10	-3	0



In this example, after three iterations, changes should stop. But in this case, change is continuing after three iterations. This is the drawback of the Bellman-Ford algorithm.

12.5.2 Single-Destination Shortest Path Problem

This type/variation of the shortest path problem finds the shortest path from all vertices to a single destination vertex. It can be reduced to a single-source shortest path algorithm by simply reversing the directions of the edges. Dijkstra's algorithm and Bellman-Ford algorithm can be applied to solve single-destination shortest path problems simply by reversing the directions of edges.

12.5.3 All-Pairs Shortest Path Problem

This type/variation of the shortest path problem finds out the shortest paths between every pair of vertices in the graph. It means if there are 3 vertices A, B, and C, then we have to find out the shortest path once by considering A as source node i.e. A-B and A-C, then by considering source vertex as B i.e. B-A and B-C, and so on. Floyd-Warshall's algorithm is the one that solves all-pairs shortest path problem.

12.5.3.1 Floyd-Warshall's Algorithm

Floyd-Warshall's algorithm is used for finding the shortest paths between all pairs of vertices in a weighted graph with positive or negative edge weights. The given graph must not contain any cycles of negative length. The main advantage of Floyd Warshall's algorithm is its simplicity. It works on the principle of matrices. The main difference between the Bellman-Ford algorithm or Dijkstra's algorithm and Floyd Warshall's algorithm is that:

- Both the Bellman-Ford algorithm and Dijkstra's algorithm only compute the shortest path from a single source.
- Floyd-Warshall's algorithm computes the shortest distances between every pair of vertices in the given graph.

Floyd-Warshall's algorithm is named in the honor of two computer scientists Robert Floyd and Stephen Warshall. Both of these scientists published this algorithm in 1962.

Algorithm-Floyd-Warshall's Algorithm

Write an algorithm that explains the Floyd-Warshall's Algorithm.

1. START
2. Declare a variable N and assign a value to it, the number of vertices of the graph.
3. Create a matrix M^0 of $N \times N$, where N is the number of vertices.
4. Fill this matrix with the weights of edges. If the edge is directed from one edge to the other then fill that cell of a matrix with the weight of the edge; otherwise, fill that cell with ∞ .
5. Declare variables K, R, and C.
6. $K = 1$
7. Repeat Step-8 to 16 WHILE ($K \leq N$)
8. Create a new Matrix M^k and copy the k^{th} row and k^{th} column of matrix M^{k-1} in M^k .
[Fill the other cells of the matrix M^k using the following formula:

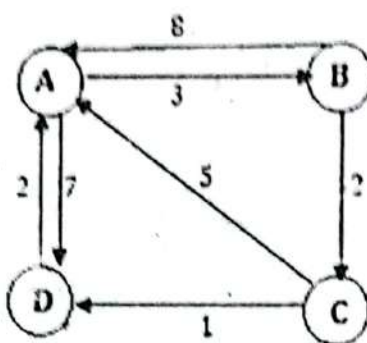
$$M^k[r, c] = \min(M^{k-1}[r, c], M^{k-1}[r, k] + M^{k-1}[k, c])$$

Where 'r' represents the row and 'c' represent the column. In this formula, 'min' is the function that returns the minimum value of two values.]
9. $R = 1$
10. Repeat Step-11 to 15 WHILE ($R \leq N$)
11. $C = 1$
12. Repeat Step-13 to 14 WHILE ($C \leq N$)
13. IF NOT ($R = K$ OR $C = K$) THEN

$$M^k[R, C] = \min(M^{k-1}[R, C], M^{k-1}[R, K] + M^{k-1}[K, C])$$
14. $C = C + 1$ [End of Inner loop of step-12]
15. $R = R + 1$ [End of upper loop of step-10]
16. $K = K + 1$ [End of loop of step-7]
17. EXIT

Example:

Consider the following directed weighted graph:



Following steps are performed for finding the shortest path using Floyd-Warshall's algorithm:

- 1- Create a matrix M^0 of 4×4 because there are 4 vertices in the given graph.
- 2- Fill this matrix with the weights of edges. If the edge is directed from one edge to the other then fill that cell of a matrix with the weight of the edge; otherwise, fill that cell with ∞ .

$$M^0 = \begin{matrix} & \begin{matrix} A & B & C & D \end{matrix} \\ \begin{matrix} A \\ B \\ C \\ D \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{bmatrix} \end{matrix}$$

- 3- Repeat this step four times (because the number of vertices of the graph are 4):

i) First Iteration:

Create a new matrix M^1 and copy the first row and first column of matrix M^0 in M^1 .

$$M^1 = \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & - & - & - \\ 5 & - & - & - \\ 2 & - & - & - \end{bmatrix}$$

Use the following formula to fill the other cells of the matrix M^1 :

$$M^1[r, c] = \min(M^0[r, c], M^0[r, k] + M^0[k, c])$$

- Value for second row and second column:

$$\begin{aligned} M^1[2, 2] &= \min(M^0[2, 2], M^0[2, 1] + M^0[1, 2]) \\ &= \min(0, 8 + 3) = \min(0, 11) = 0 \end{aligned}$$

- Value for second row and third column:

$$M^1[2, 3] = \min(M^0[2, 3], M^0[2, 1] + M^0[1, 3])$$

$$= \min(2, 8 + \infty) = \min(2, \infty) = 2$$

- Value for second row and fourth column:

$$M^1[2, 4] = \min(M^0[2, 4], M^0[2, 1] + M^0[1, 4])$$

$$= \min(\infty, 8 + 7) = \min(\infty, 15) = 15$$

- Value for third row and second column:

$$M^1[3, 2] = \min(M^0[3, 2], M^0[3, 1] + M^0[1, 2])$$

$$= \min(\infty, 5 + 3) = \min(\infty, 8) = 8$$

- Value for third row and third column:

$$M^1[3, 3] = \min(M^0[3, 3], M^0[3, 1] + M^0[1, 3])$$

$$= \min(0, 5 + \infty) = \min(0, \infty) = 0$$

- Value for third row and fourth column:

$$M^1[3, 4] = \min(M^0[3, 4], M^0[3, 1] + M^0[1, 4])$$

$$= \min(1, 5 + 7) = \min(0, 12) = 1$$

- Value for fourth row and second column:

$$M^1[4, 2] = \min(M^0[4, 2], M^0[4, 1] + M^0[1, 2])$$

$$= \min(\infty, 2 + 3) = \min(\infty, 5) = 5$$

- Value for fourth row and third column:

$$M^1[4, 3] = \min(M^0[4, 3], M^0[4, 1] + M^0[1, 3])$$

$$= \min(\infty, 2 + \infty) = \min(\infty, \infty) = \infty$$

- Value for fourth row and fourth column:

$$M^1[4, 4] = \min(M^0[4, 4], M^0[4, 1] + M^0[1, 4])$$

$$= \min(0, 2 + 7) = \min(0, 9) = 0$$

Thus the matrix M^1 is:

$$M^1 = \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & \infty & 0 \end{bmatrix}$$

ii) Second Iteration:

Create a new matrix M^2 and copy the second row and second column of matrix M^1 in M^2 .

$$M^2 = \begin{bmatrix} - & 3 & - & - \\ 8 & 0 & 2 & 15 \\ - & 8 & - & - \\ - & 5 & - & - \end{bmatrix}$$

Use the following formula to fill the other cells of the matrix M^2 :

$$M^2[r, c] = \min(M^1[r, c], M^1[r, k] + M^1[k, c])$$

- Value for first row and first column:

$$\begin{aligned} M^2[1, 1] &= \min(M^1[1, 1], M^1[1, 2] + M^1[2, 1]) \\ &= \min(0, 3+8) = \min(0, 11) = 0 \end{aligned}$$

- Value for first row and third column:

$$\begin{aligned} M^2[1, 3] &= \min(M^1[1, 3], M^1[1, 2] + M^1[2, 3]) \\ &= \min(\infty, 3+2) = \min(\infty, 5) = 5 \end{aligned}$$

- Value for first row and fourth column:

$$\begin{aligned} M^2[1, 4] &= \min(M^1[1, 4], M^1[1, 2] + M^1[2, 4]) \\ &= \min(7, 3+15) = \min(7, 18) = 7 \end{aligned}$$

- Value for third row and first column:

$$\begin{aligned} M^2[3, 1] &= \min(M^1[3, 1], M^1[3, 2] + M^1[2, 1]) \\ &= \min(5, 8+8) = \min(5, 16) = 5 \end{aligned}$$

- Value for third row and third column:

$$\begin{aligned} M^2[3, 3] &= \min(M^1[3, 3], M^1[3, 2] + M^1[2, 3]) \\ &= \min(0, 8+2) = \min(0, 10) = 0 \end{aligned}$$

- Value for third row and fourth column:

$$\begin{aligned} M^2[3, 4] &= \min(M^1[3, 4], M^1[3, 2] + M^1[2, 4]) \\ &= \min(1, 8+15) = \min(1, 23) = 1 \end{aligned}$$

- Value for fourth row and first column:

$$\begin{aligned} M^2[4, 1] &= \min(M^1[4, 1], M^1[4, 2] + M^1[2, 1]) \\ &= \min(2, 5+8) = \min(2, 13) = 2 \end{aligned}$$

- Value for fourth row and third column:

$$\begin{aligned} M^2[4, 3] &= \min(M^1[4, 3], M^1[4, 2] + M^1[2, 3]) \\ &= \min(\infty, 5+2) = \min(\infty, 7) = 7 \end{aligned}$$

- Value for fourth row and fourth column:

$$\begin{aligned} M^2[4, 4] &= \min(M^1[4, 4], M^1[4, 2] + M^1[2, 4]) \\ &= \min(0, 5+15) = \min(0, 20) = 0 \end{aligned}$$

Thus the matrix M^2 is:

$$M^2 = \begin{bmatrix} 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

III) Third Iteration:

Create a new matrix M^3 and copy the third row and third column of M^2 as it is in M^3 .

$$M^3 = \begin{bmatrix} - & - & 5 & - \\ - & - & 2 & - \\ 5 & 8 & 0 & 1 \\ - & - & 7 & - \end{bmatrix}$$

Use the following formula to fill the other cells of the matrix M^3 :

$$M^3[r, c] = \min(M^2[r, c], M^2[r, k] + M^2[k, c])$$

- Value for first row and first column:

$$\begin{aligned} M^3[1, 1] &= \min(M^2[1, 1], M^2[1, 3] + M^2[3, 1]) \\ &= \min(0, 5+5) = \min(0, 10) = 0 \end{aligned}$$

- Value for first row and second column:

$$\begin{aligned} M^3[1, 2] &= \min(M^2[1, 2], M^2[1, 3] + M^2[3, 2]) \\ &= \min(3, 5+8) = \min(3, 13) = 3 \end{aligned}$$

- Value for first row and fourth column:

$$\begin{aligned} M^3[1, 4] &= \min(M^2[1, 4], M^2[1, 3] + M^2[3, 4]) \\ &= \min(7, 5+1) = \min(7, 6) = 6 \end{aligned}$$

- Value for second row and first column:

$$\begin{aligned} M^3[2, 1] &= \min(M^2[2, 1], M^2[2, 3] + M^2[3, 1]) \\ &= \min(8, 2+5) = \min(8, 7) = 7 \end{aligned}$$

- Value for second row and second column:

$$\begin{aligned} M^3[2, 2] &= \min(M^2[2, 2], M^2[2, 3] + M^2[3, 2]) \\ &= \min(0, 2+8) = \min(0, 10) = 0 \end{aligned}$$

- Value for second row and fourth column:

$$\begin{aligned} M^3[2, 4] &= \min(M^2[2, 4], M^2[2, 3] + M^2[3, 4]) \\ &= \min(15, 2+1) = \min(15, 3) = 3 \end{aligned}$$

- Value for fourth row and first column:

$$\begin{aligned} M^3[4, 1] &= \min(M^2[4, 1], M^2[4, 3] + M^2[3, 1]) \\ &= \min(2, 7+5) = \min(2, 12) = 2 \end{aligned}$$

- Value for fourth row and second column:

$$\begin{aligned} M^3[4, 2] &= \min(M^2[4, 2], M^2[4, 3] + M^2[3, 2]) \\ &= \min(5, 7+8) = \min(5, 15) = 5 \end{aligned}$$

- Value for fourth row and fourth column:

$$\begin{aligned} M^3[4, 4] &= \min(M^2[4, 4], M^2[4, 3] + M^2[3, 4]) \\ &= \min(0, 7+1) = \min(0, 8) = 0 \end{aligned}$$

Thus the matrix M^3 is:

$$M^3 = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 7 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

iv) Fourth Iteration:

Create a new matrix M^4 and copy the fourth row and fourth column of M^3 as it is to M^4 .

$$M^4 = \begin{bmatrix} - & - & - & 6 \\ - & - & - & 3 \\ - & - & - & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

Use the following formula to fill the other cells of the matrix M^4 :

$$M^4[r, c] = \min(M^3[r, c], M^3[r, k] + M^3[k, c])$$

- Value for first row and first column:

$$\begin{aligned} M^4[1, 1] &= \min(M^3[1, 1], M^3[1, 4] + M^3[4, 1]) \\ &= \min(0, 6+2) = \min(0, 8) = 0 \end{aligned}$$

- Value for first row and second column:

$$\begin{aligned} M^4[1, 2] &= \min(M^3[1, 2], M^3[1, 4] + M^3[4, 2]) \\ &= \min(3, 6+5) = \min(3, 11) = 3 \end{aligned}$$

- Value for first row and third column:

$$\begin{aligned} M^4[1, 3] &= \min(M^3[1, 3], M^3[1, 4] + M^3[4, 3]) \\ &= \min(5, 6+7) = \min(5, 13) = 5 \end{aligned}$$

- Value for second row and first column:

$$\begin{aligned} M^4[2, 1] &= \min(M^3[2, 1], M^3[2, 4] + M^3[4, 1]) \\ &= \min(7, 3+2) = \min(7, 5) = 5 \end{aligned}$$

- Value for second row and second column:

$$\begin{aligned} M^4[2, 2] &= \min(M^3[2, 2], M^3[2, 4] + M^3[4, 2]) \\ &= \min(0, 3+5) = \min(0, 8) = 0 \end{aligned}$$

- Value for second row and third column:

$$\begin{aligned} M^4[2, 3] &= \min(M^3[2, 3], M^3[2, 4] + M^3[4, 3]) \\ &= \min(2, 3+7) = \min(2, 10) = 2 \end{aligned}$$

- Value for third row and first column:

$$\begin{aligned} M^4[3, 1] &= \min(M^3[3, 1], M^3[3, 4] + M^3[4, 1]) \\ &= \min(5, 1+2) = \min(5, 3) = 3 \end{aligned}$$

- Value for third row and second column:

$$\begin{aligned} M^4[3, 2] &= \min(M^3[3, 2], M^3[3, 4] + M^3[4, 2]) \\ &= \min(8, 1+5) = \min(8, 6) = 6 \end{aligned}$$

- Value for third row and third column:

$$\begin{aligned} M^4[3, 3] &= \min(M^3[3, 3], M^3[3, 4] + M^3[4, 3]) \\ &= \min(0, 1+7) = \min(0, 8) = 0 \end{aligned}$$

The matrix M^4 is:

$$M^4 = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 3 & 6 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

Thus, the shortest distance between each pair can be determined by the matrix M^4 that is:

	A	B	C	D
A	0	3	5	6
B	5	0	2	3
C	3	6	0	1
D	2	5	7	0

Time Complexity of Floyd-Warshall's Algorithm

The time complexity of Floyd-Warshall's algorithm is $O(V^3)$, where V is the number of vertices in the graph.

Program Code 12.4

// This program implements the above example of Floyd-Warshall's algorithm.

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
class Graph
```

```
{
```

```
    public:
```

```
        // This function processes matrix in different iterations
```

```
        // Two matrices temp1 and temp2 are used as arguments
```

```
        // Temp1 is used as input while temp2 is used as output.
```

```
        // We know that data is passed to the matrix as a reference, not by value
```



```

void Process_Matrix (int temp1[4][4], int temp2[4][4], int K)
{
    int R, C;
    // This code copies values of row K and column K of temp1
    // as it is, to temp2. Other cells of temp2 are initialized with 0 value

    for (R = 0; R<=3;R++)
    {
        for (C = 0; C<=3;C++)
        {
            if((R ==K || C == K))
                temp2[R][C] = temp1[R][C];
            else
                temp2[R][C] = 0;
        }
    }

    // This code fills other cells of matrix temp2 using the formula
    //  $M^k[r, c] = \min(M^{k-1}[r, c], M^{k-1}[r, k] + M^{k-1}[k, c])$  and function 'min'
    // which is used-defined function

    for (R = 0; R<=3;R++)
    {
        for (C = 0; C<=3;C++)
        {
            if(!(R ==K || C == K))
                temp2[R][C] = min(temp1[R][C], temp1[R][K] + temp1[K][C]);
        }
    }

    // This function finds out the minimum value from two values
    int min(int x, int y)
    {
        if(x>= 999 && y>=999)
            return 999;
        else if(x>= 999)
            return y;
        else if(y>= 999)
            return x;
        else

```

```

        return (x<y)?x:y;
    }

    // This function prints the values of all elements of the given matrix
    void print_matrix(int M[4][4])
    {
        clrscr();
        for (int r = 0; r<4; r++)
        {
            for (int c = 0; c<4; c++)
                cout<<M [r][c]<<"\t";
            cout<<endl;
        }
    }
};

void main()
{
    Graph g;

    // Matrix M0 is declared and initialized with values of the matrix as mentioned
    // in the above example. Value 999 is stored in place of  $\infty$  (infinity)
    int M0[4][4] = {{0,3,999,7}, {8,0,2, 999},{5,999,0,1},{2,999,999,0}};
    //four matrices are declared for using in different iterations
    int M1[4][4], M2[4][4], M3[4][4], M4[4][4];
    clrscr();

    // First Iteration: Addresses of matrix M0 and M1 are passed with a function call
    g.Process_Matrix(M0, M1, 0);
    // Second Iteration: Addresses of matrix M1 and M2 are passed with a function call
    g.Process_Matrix(M1, M2, 1);
    // Third Iteration: Addresses of matrix M2 and M3 are passed with a function call
    g.Process_Matrix(M2, M3, 2);
    // Fourth Iteration: Addresses of matrix M3 and M4 are passed with a function call
    g.Process_Matrix(M3, M4, 3);
    // Printing the values of resultant matrix M4
    g.print_matrix(M4);
    getch();
}

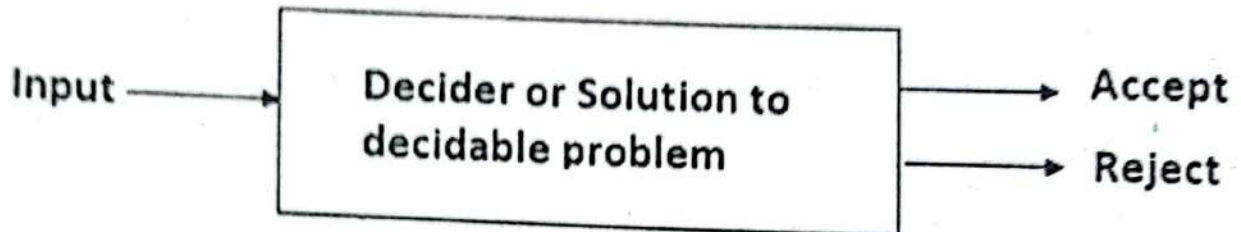
```

Output of the program:

0	3	5	6
5	0	2	3
3	6	0	1
2	5	7	0

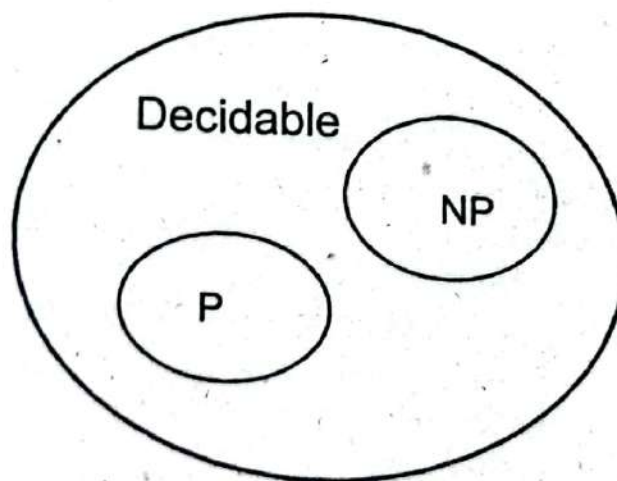
12.6 NP-COMPLETENESS

We have seen different problems and related algorithms to solve these problems. All these problems are decidable problems (or decision problems). Decidable problems are those that can be computationally solved by a computer. They always give output either positive or negative. That is either it accepts the input or rejects; no other option. They do not hang on a loop forever or halts for some input in between the processing. The block diagram for decidable algorithms is as follows:



All the algorithms discussed in this book are for decidable problems. They must answer. Un-decidable problems are opposite to it. They halt in the middle of processing or stuck in an infinite loop and the machine is not able to recognize easily. There is no mechanism that can tell about the problem. They are also known as *hard problems* because sometimes they might have a hard time for checking them. The halting problem is one of the un-decidable problems.

Decidable problems are further divided into P problems and NP problems based on the running times of algorithms. Following figure shows the relationship between P and NP problems:



12.6.1 P Problems

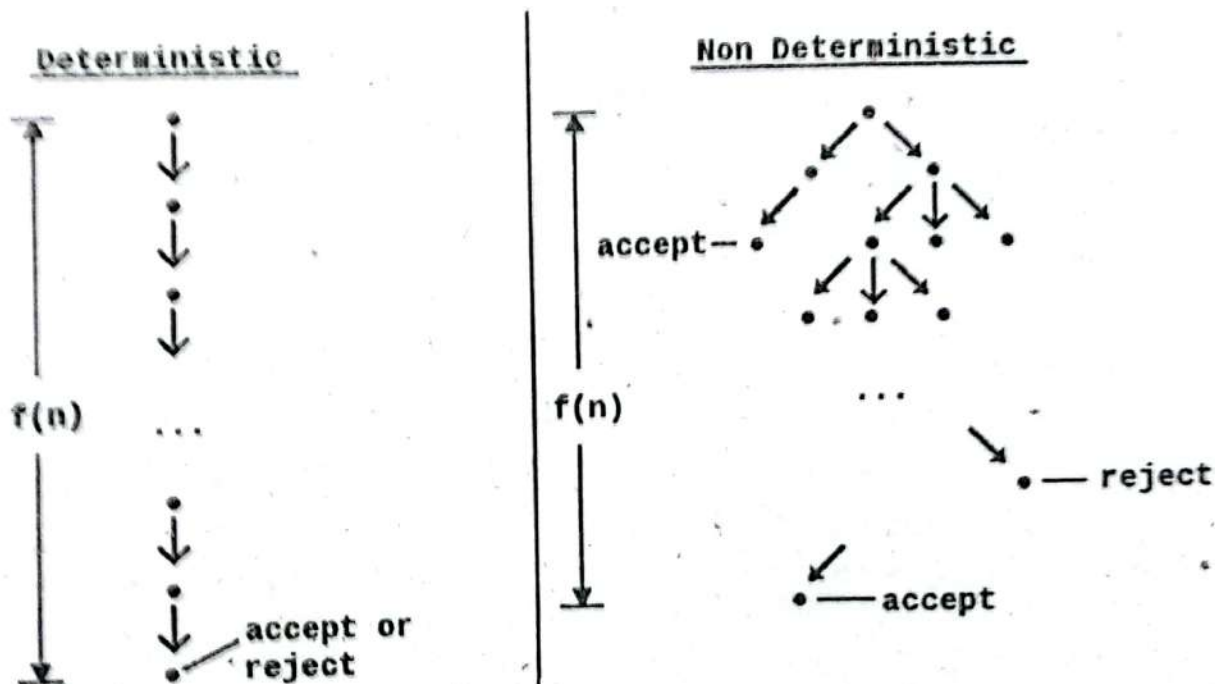
P stands for polynomial. The P class problem is the set of all problems that can be solved with worst-case polynomial time complexity. In other words, a problem belongs to class P if it is a decision problem and there exists an algorithm that solves any instance in polynomial time. The polynomial time is expressed as:

$$P = \bigcup_{\infty} \text{TIME}(n^k)$$

P problems are *tractable problems*. Tractable problems are those that are quickly checkable and quickly solvable. It means that these problems are solvable in theory as well as in practice. All the algorithms that we have been discussed having polynomial time complexity are P-class problems like Floyd-Warshall's algorithm, Bellman-Ford algorithm, and Binary sort.

12.6.2 NP Problems

NP stands for Nondeterministic Polynomial. A problem is called NP if its solution can be guessed and verified in polynomial time. The word 'non-deterministic' means that no particular rule is followed to make the guess. The algorithm shows different behaviors on different runs with the same input.



A simple way to check if a problem belongs to NP class is to phrase the problem to yes/no question. The problem is in NP if, in polynomial time, any of the 'yes' instance is correct. No need to worry about 'no' instances.

NP problems are Interactable problems. Interactable problems are quickly checkable but not quickly solvable. It means they are solvable in theory but not in practice. In practice, some of its instances may take centuries to solve.

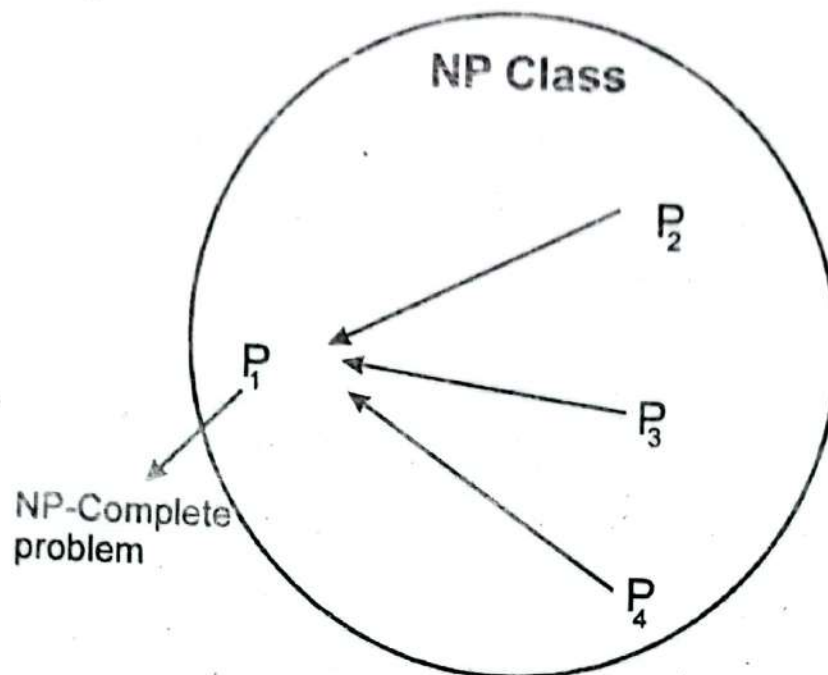
NP class problems are exponential time algorithms, and are expressed as:

$$NP = \bigcup_{k \geq 0} TIME(2^{n^k})$$

Some of the examples of NP class problems are Hamiltonian path algorithms, Clique problem, Satisfiability problem, and subset-sum problem.

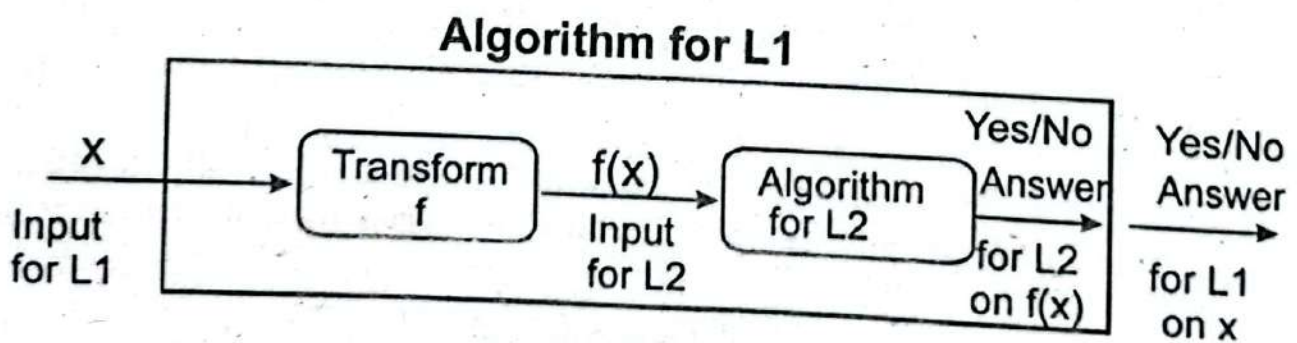
12.6.3 NP-Completeness

Among all the problems known to be in NP class, there is a subset, known as the *NP-complete problems*, that has the property that any problem in NP can be reduced to it in polynomial time.



In the above figure, suppose NP class problems are P_1 , P_2 , P_3 , and P_4 . If P_2 , P_3 , and P_4 are reduced to P_1 in polynomial time, then P_1 is said to be the NP-complete problem.

Reduction means to transform an algorithm to design another algorithm. For example, let L_1 and L_2 be two decision problems. Suppose there is an algorithm for problem L_2 that answers Yes or No depending upon input. The algorithm for problem L_1 can be designed using the algorithm of L_2 . The idea is to find a transformation so that the algorithm of L_2 can be part of an algorithm of L_1 to solve L_1 . The block diagram of the algorithm L_1 by reducing algorithm L_2 is shown in the following figure:



In complexity theory, the NP-complete problems are the most difficult in NP, in the sense that they are the ones most likely not to be in P class. If one could find a way to solve any NP-complete problem quickly in polynomial time, then they could use that algorithm to solve all NP problems quickly. NP-complete problems are often addressed by approximation, probabilistic, and heuristic approaches. Some well-known problems that are NP-complete are as follows:

- Boolean satisfiability problem (SAT)
- N-puzzle
- Knapsack problem
- Hamiltonian cycle problem
- Traveling salesman problem
- Subgraph isomorphism problem
- Subset sum problem
- Clique problem
- Vertex cover problem
- Independent set problem
- Graph coloring problem
- Minesweeper
- Bin packing

Source Code of Programs

Source code of programs given in "DATA STRUCTURES USING C++" can be received by sending an e-mail to "pakman_series@hotmail.com".

Short Answers to the Questions

 **What is the difference between vertices and edges of a graph?**

The individual data elements (or nodes) of a graph are called *vertices*, whereas lines that connect vertices are called *edges*.

 **What is the difference between directed and undirected graph?**

A graph that has only directed edges from one vertex to another is called a *directed graph*, whereas a graph that has only undirected edges is called an *undirected graph*.

 **What do you know about cycle and acyclic?**

A path that originates or begins and ends at the same vertex is called a *cycle*, whereas the directed graph that has no cycles is called an *acyclic graph*.

 **What is the difference between a simple graph and a complete graph?**

A graph is said to be a simple graph if it does not contain any loop-edge (or self-loop), whereas a graph in which all pairs of vertices are adjacent is called a *complete graph*.

What is the difference between adjacent vertices and adjacent edges?

Two vertices are adjacent if they are connected to each other by an edge, whereas two edges are adjacent if they share the same vertex.

What is the use of Floyd-Warshall's algorithm?

Floyd-Warshall's algorithm is used for finding the shortest paths between all pairs of vertices in a weighted graph with positive or negative edge weights. The given graph must not contain any cycles of negative length. The main advantage of Floyd Warshall's algorithm is its simplicity. It works on the principle of matrices.

What is the difference between the Bellman-Ford algorithm or Dijkstra's algorithm and Floyd-Warshall's algorithm?

The main difference between the Bellman-Ford algorithm or Dijkstra's algorithm and Floyd Warshall's algorithm is that:

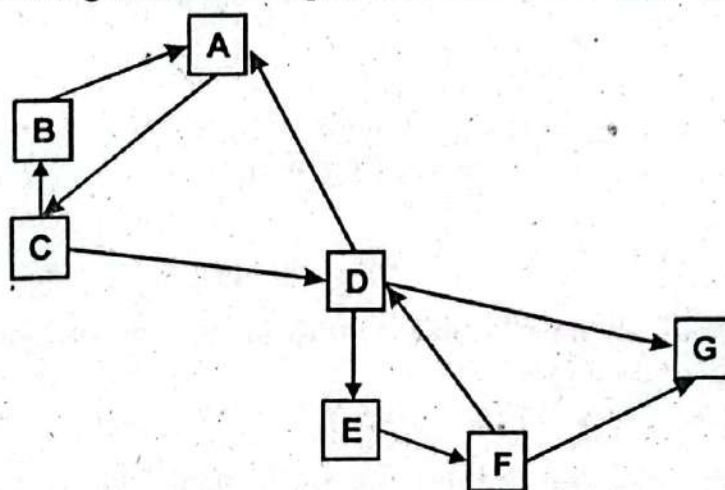
- Both the Bellman-Ford algorithm and Dijkstra's algorithm only compute the shortest path from a single source.
- Floyd-Warshall's algorithm computes the shortest distances between every pair of vertices in the given graph.

EXERCISE

Q# 1 Select the correct answer from the multiple choices.

- It is a non-linear data structure:
(a) Linked List (b) Array (c) Stack (d) Graph
- In a graph, lines that connect vertices are called:
(a) Vertices (b) Nodes (c) Edges (d) Degree
- A directed graph is also called:
(a) Pigraph (b) Digraph (c) Directional graph (d) Bigraph
- A graph with 'n' vertices will definitely have a parallel edge or a self loop if the total number of edges is:
(a) greater than $n - 1$
(b) less than $n(n - 1)$
(c) greater than $n(n - 1)/2$
(d) less than $n^2/2$
- Where can we use Breadth First Search(BFS)?
(a) Stacks (b) Strings (c) Graphs (d) Binary Tree
- How many edges are present in a complete graph with 'n' vertices?
(a) $n*(n - 1)$
(b) $n*(n - 1) / 2$
(c) $n*(n - 2)$
(d) $(n - 1) / 2$
- Which of the following graph element represents a simple path that begins and terminates at the same vertex?
(a) path (b) arc (c) Edge (d) cycle

8. Which data structure is used in Breadth First Search of a graph to hold nodes?
 (a) Stack (b) Queue (c) Tree (d) Array
9. A directed graph is _____ if there is a path from each vertex to every other vertex in the digraph.
 (a) Weakly connected (b) Strongly Connected
 (c) Tightly Connected (d) Linearly Connected
10. In the _____ traversal we process all of a vertex's descendants before we move to an adjacent vertex.
 (a) Depth First (b) Breadth First (c) Depth Limited (d) With First
11. A complete graph can have:
 (a) n^2 spanning trees (b) $n^{(n-2)}$ spanning trees
 (c) $n^{(n+1)}$ spanning trees (d) n^n spanning trees
12. Graphs are represented using:
 (a) Adjacency tree (b) Adjacency linked list
 (c) Adjacency graph (d) Adjacency queue
13. A graph is said to be _____ if every node u in G is adjacent to every other node v in G .
 (a) Absolute (b) Full (c) Inclusive (d) Complete
14. A connected graph T without any cycles is called:
 (a) a tree graph (b) free tree (c) a tree (d) All
15. In a graph if $e = [u, v]$, then u and v are called:
 (a) endpoints of e (b) adjacent vertices (c) neighbors (d) All
16. In a graph if $e = (u, v)$ means:
 (a) u is adjacent to v but v is not adjacent to u . (b) e begins at u and ends at v
 (c) u is node and v is an edge (d) both u and v are edges.
17. Which of the following is an invalid path for the following graph?



- (a) C B A (b) C D E F G (c) C D A B C A (d) C D A
18. An undirected graph G with n vertices and e edges is represented by an adjacency list. What is the time required to generate all the connected components?
 (a) $O(n)$ (b) $O(e)$ (c) $O(e+n)$ (d) $O(e^2)$

19. The maximum degree of any vertex in a simple graph with n vertices is:
(a) $n-1$ (b) $n+1$ (c) $2n-1$ (d) n
20. For an undirected graph with n vertices and e edges, the sum of the degree of each vertex is equal to:
(a) $2n$ (b) $(2n-1)/2$ (c) $2e$ (d) $(e^2)/2$
21. The spanning tree of connected graph with 10 vertices contains:
(a) 9 edges (b) 11 edges (c) 10 edges (d) 9 vertices
22. Which of the following algorithms solves the all-pair shortest path problem?
(a) Floyd Warshall's algorithm (b) Prim's algorithm
(c) Dijkstra's algorithm (d) Bellman Ford algorithm
23. Which of the following is useful in traversing a given graph by breadth first search?
(a) Set (b) List (c) Stack (d) Queue
24. The minimum number of edges in a connected cyclic graph on n vertices is:
(a) $n-1$ (b) n (c) $n+1$ (d) None
25. _____ is known as a greedy algorithm, because it chooses at each step the cheapest edge to add to sub-graph S .
(a) Kruskal's algorithm (b) Prim's algorithm
(c) Dijkstra algorithm (d) Both a & b
26. Dijkstra algorithm is also called the _____ shortest path problem.
(a) multiple source (b) single source
(c) single destination (d) multiple destination
27. _____ solves the problem of finding the shortest path from a point in a graph to a destination.
(a) Kruskal's algorithm (b) Prim's algorithm
(c) Dijkstra algorithm (d) Bellman Ford algorithm
28. _____ is a most generalized single source shortest path algorithm to find the shortest path in a graph even with negative weights.
(a) Kruskal's algorithm (b) Prim's algorithm
(c) Dijkstra algorithm (d) Bellman Ford algorithm
29. The Floyd-Warshall all pairs shortest path algorithm computes the shortest paths between each pair of nodes in:
(a) $O(\log n)$ (b) $O(n^2)$ (c) $O(mn)$ (d) $O(n^3)$
30. The _____ process updates the costs of all the vertices V , connected to a vertex U , if we could improve the best estimate of the shortest path to V by including (U,V) in the path to V .
(a) relaxation (b) improvement (c) Shortening (d) Costing

Answers of MCQs

01 (d)	02 (c)	03 (b)	04 (a)	05 (c)	06 (b)	07 (d)	08 (b)	09 (b)	10 (a)
11 (b)	12 (b)	13 (d)	14 (d)	15 (d)	16 (b)	17 (c)	18 (c)	19 (a)	20 (c)
21 (a)	22 (a)	23 (d)	24 (b)	25 (d)	26 (b)	27 (c)	28 (d)	29 (d)	30 (a)

Q.2 What is a graph in the data structure? Explain its applications in real-world systems.

Q.3 How a directed graph differs from a tree?

Q.4 Explain different methods for graph representations.

Q.5 What is meant by graph traversing? Explain different ways to traverse a graph.

Q.6 Differentiate between the following:

- i) in-degree & out-degree of a vertex
- ii) Parallel edges and loop edges
- iii) Sparse graph and dense graph

Q.7 Define the spanning tree and its properties. Explain the minimum spanning tree with an example.

Q.8 Which algorithms are used for finding the minimum spanning tree from an input graph? Briefly explain these algorithms.

Q.9 Explain the shortest path problem and its applications.

Q.10 Explain the Dijkstra's algorithm with an example.

Q.11 What do you know about P problems and NP problems?

Complexities of Common Operations of Data Structures

Linear Array

Operation	Time Complexity		Space Complexity
	Average-Case	Worst-Case	Worst-Case
Access	$\Theta(1)$	$O(1)$	$O(n)$
Traversal	$\Theta(n)$	$O(n)$	$O(n)$
Search	$\Theta(n)$	$O(n)$	$O(n)$
Insertion	$\Theta(n)$	$O(n)$	$O(n)$
Deletion	$\Theta(n)$	$O(n)$	$O(n)$

Stack

Operation	Time Complexity		Space Complexity
	Average-Case	Worst-Case	Worst-Case
Pushing (Insertion)	$\Theta(1)$	$O(1)$	$O(n)$
Popping (Deletion)	$\Theta(1)$	$O(1)$	$O(n)$

Queue

Operation	Time Complexity		Space Complexity
	Average-Case	Worst-Case	Worst-Case
Insertion	$\Theta(1)$	$O(1)$	$O(n)$
Deletion	$\Theta(1)$	$O(1)$	$O(n)$
Access	$\Theta(n)$	$O(n)$	$O(n)$
Search	$\Theta(n)$	$O(n)$	$O(n)$

Sorting Algorithms

Sorting Algorithm	Time Complexity			Space Complexity
	Best-Case	Average-Case	Worst-Case	Worst-Case
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Shell Sort	$\Omega(n \log n)$	$\Theta(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$
Quick Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	$O(\log n)$
Merge Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$
Counting Sort	$\Omega(n + k)$	$\Theta(n + k)$	$O(n + k)$	$O(n + k)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n + k)$
Bucket Sort	$\Omega(n + k)$	$\Theta(n + k)$	$O(n^2)$	$O(n + k)$

Searching Algorithms

Search Algorithm	Time Complexity		Space Complexity
	Average-Case	Worst-Case	Worst-Case
Sequential Search	$\Theta(\frac{n+1}{2}) \approx \Theta(n)$	$O(n)$	$O(n)$
Binary Search	$\Theta(\log n)$	$O(\log n)$	$O(n)$

Hash Table

Operation	Time Complexity		Space Complexity
	Average-Case	Worst-Case	Worst-Case
Search	$\Theta(1)$	$O(n)$	$O(n)$
Insertion	$\Theta(1)$	$O(n)$	$O(n)$
Deletion	$\Theta(1)$	$O(n)$	$O(n)$

Linked List

Linked List	Time Complexity								Space Complexity
	Average-Case				Worst-Case				Worst-Case
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Singly Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Doubly Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Trees

Tree	Time Complexity								Space Complexity
	Average-Case				Worst-Case				Worst-Case
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Binary Search Tree (BST)	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
AVL	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Red-Black Tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Splay Tree	-	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	-	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
B-Tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$

PM SERIES BOOKS



www.youtube.com/PMSeriesSupport



www.facebook.com/PMSeries.Support

Books for BS(CS)/ADP/B.Sc/MCS/M.Sc

- THEORY OF AUTOMATA
- DATA STRUCTURES USING C++
- INTRODUCTION TO COMPUTER INFORMATION TECHNOLOGY
- OPERATING SYSTEMS
- DATABASE MANAGEMENT
- VISUAL BASIC 6.0
- WEB PROGRAMMING
- ADVANCED PROGRAMMING WITH C++
- SOFTWARE ENGINEERING

Books for Intermediate (ICS)

- COMPUTER SCIENCE
(for ICS Part-1 — Punjab Board)
- COMPUTER SCIENCE
(for ICS Part-2 — Punjab Board)
- Practical Notebook COMPUTER SCIENCE
(for ICS Part-1 & 2 — Punjab Board)
- COMPUTER SCIENCE
(for ICS Part-1 — Federal Board)
- COMPUTER SCIENCE
(for ICS Part-2 — Federal Board)

Books for DAE

- COMPUTER APPLICATIONS
(for COMP - 122 — Urdu Medium)
- Practical Notebook
COMPUTER APPLICATIONS
(for COMP - 122 — Urdu Medium)
- Practical Notebook
COMPUTER APPLICATIONS
(for COMP - 122 — English Medium)

Books for AIU

- I.C.T
(for B.A. — Course Code 1431)
- BASICS OF I.C.T
(for B.Ed — Course Code 5403)

Books for Commerce

- COMPUTER APPLICATIONS IN BUSINESS
(for B.Com)
- COMPUTER STUDIES
(for I.Com)
- BUSINESS INFORMATION TECHNOLOGY
(for D.Com Part-1 — Urdu Medium)
- BUSINESS INFORMATION TECHNOLOGY
(for D.Com Part-2 — Urdu Medium)

Books for B.Ed

- INFORMATION COMMUNICATION TECHNOLOGY IN EDUCATION
(for B.Ed Secondary & Elementary)
- COMPUTER IN EDUCATION (English Medium)
(for B.Ed Sargodha University — Code 504)
- COMPUTER IN EDUCATION (Urdu Medium)
(for B.Ed Sargodha University — Code 504)

Books for MATRIC

- COMPUTER SCIENCE
(for 9th Class — Urdu Medium)
- COMPUTER SCIENCE
(for 9th Class — English Medium with Urdu Translation)
- COMPUTER SCIENCE
(for 9th Class — English Medium)
- COMPUTER SCIENCE
(for 10th Class — Urdu Medium)
- COMPUTER SCIENCE
(for 10th Class — English Medium with Urdu Translation)
- COMPUTER SCIENCE
(for 10th Class — English Medium)

Books for Middle

- COMPUTER EDUCATION
(for 6th, 7th, 8th Classes — Urdu Medium)
- COMPUTER EDUCATION
(for 6th, 7th, 8th Classes — English Medium with Urdu Translation)



**MAJEED
SONS**

Head Office

22-Urdu Bazar, Lahore

Ph:042-37311484, 37355187

Aminpur Bazar, Faisalabad. Ph: 041-2643322

Al-Mustafa Plaza, 6th Road, Rawalpindi Ph: 051-4423948



www.majeedbookdepot.com



majeedbookdepot@gmail.com



[majeedbookdepot](https://www.facebook.com/majeedbookdepot)